Thwarting Advanced Code-reuse Attacks

by

Misiker Tadesse Aga

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2020

Doctoral Committee:

Professor Todd M. Austin, Chair
Assistant Professor Jean-Baptiste Jeannin
Assistant Professor Baris Can Cengiz Kasikci
Professor Scott Mahlke

Misiker Tadesse Aga

misiker@umich.edu

ORCID iD: 0000-0002-8084-6334

To my family

Acknowledgments

First and foremost, I would like express my deepest gratitude to my advisor Professor Todd Austin for his unwavering support and great mentorship both professionally and personally. Todd has been a great source of inspiration and taught me a lot about how to do research and how to effectively communicate my work. I would also like to thank my thesis committee members Professor Jean-Baptiste Jeannin, Professor Baris Kasikci and Professor Scott Mahlke for their support and valuable feedback. Thank you Professor Valeria Bertacco for your valuable feedback on my research and presentations.

I owe a special thanks to my collaborators. Thank you Colton Holoday for doing a lot of heavy lifting during the implementation of ProxyCFI. Thank you Zelalem Aweke and Salessawi Ferede for your collaborative efforts in our joint projects.

I would like to thank my colleagues at the University of Michigan including Abraham Addisie, Biruk Mammo, William Arthur, Zelalem Aweke, Salessawi Ferede for all the technical discussions, suggestions and brainstorming ideas. I would also like to thank past and present AB-Research group members including Lauren Biernacki, Mark Gallagher, Doowon Lee, Pete Ehrett, Brendan West, Shibo Chen, Taru Verma, Vidushi Goyal, Timothy Linscott, Andrew McCrabb, Hiwot Kassa and Leul Belayneh for the fun discussions, for reviewing my papers and slide decks. I also like to thank Annika Pattenaude for helping me with writing edits and reviews. Thank you all my friends in Ann Arbor - Abraham Addisie, Leul Belayneh, Kidus Admassu, Zelalem Aweke, Salessawi Ferede, Fitsum Andargie and a lot more others that I haven't mentioned here for making my stay entertaining.

I also like to thank the CSE staff who were always helpful and made everything run smoothly during my stay at the University of Michigan. Thank you Ashley Andreae, Stephen Reger, Karen Liska, Christine Boltz, Alice Melloni, Tracie Straub, Laura Pasek, Denise DuPrie, Erika Hauffman and a whole lot others.

I am indebted to my family for supporting me throughout this endeavors. My parents, Almaz Bekele and Tadesse Aga, for always being there for me, constantly supporting and encouraging me to pursue my studies. I thank my siblings Dagmawit, Hadlawit, Hamrawit, Yared and Kaleab for their constant support and for guiding me through important decisions

in my life. Finally, I would like to express my gratitude to my fiancé Fikirte for always believing in me and keeping me strong throughout this endeavors.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Code-reuse attacks are the leading mechanism by which attackers infiltrate systems. Various mitigation techniques have been proposed to defend against these attacks, the most prominent one being control-flow integrity (CFI). CFI is a principled approach that restricts all indirect control flows to adhere to a statically determined control-flow graph (CFG). CFI has gained widespread adoption in industry – such as Microsoft Control Flow Guard and Intel Control-flow Enforcement Technology. However, recent attacks dubbed CFG mimicry attacks, like control flow bending and counterfeit object-oriented programming, have shown that code-reuse attacks are still possible without violating CFI. Furthermore, data-oriented programming (DOP) has generalized non-control data attacks to achieve Turing-complete computation; it accomplishes this by repeatedly corrupting non-control data to execute a sequence of instructions within the legitimate control flow of the program. In this dissertation, we present techniques to mitigate these advanced code-reuse attacks.

First, this dissertation presents a novel approach to thwart advanced control flow attacks called ProxyCFI. ProxyCFI replaces all code pointers in a program with a less powerful construct: pointer proxies. Pointer proxies are random identifiers associated with each legitimate control flow edge in the program. Pointer proxy values are defined per-function and are re-randomized at program load time to mitigate their disclosure. To ensure that the approach covers the entire control flow of the program, we have a load-time verifier built-in the program loader that performs reachability analyses of the code and verify that there is no vulnerable control flow transfer. ProxyCFI delivers these protections incurring minimal performance overhead, while stopping a broad range of real-world attacks and achieving a 100% coverage of the RIPE x86-64 attack suite.

Second, this dissertation evaluates the effectiveness of previously proposed stack layout randomization techniques against attacks that only utilize relative offset between allocations (e.g., data-oriented programming) and demonstrate that they are ineffective at stopping real-world DOP exploits. We then propose Smokestack, a runtime stack-layout randomization technique that addresses the problems with prior approaches. Smokestack instruments programs to randomize their stack layout at runtime for each invocation of a function. By

doing so, Smokestack minimizes the utility of information gained in the probes of chained DOP attacks for later attack stages. Our evaluation on SPEC benchmarks and various real-world applications shows that Smokestack, with a cryptographically secure pseudo random generator, can stop DOP attacks with an average slowdown of 8.7%.

Lastly, we present a technique to randomize heap allocations at runtime to prevent attackers from orchestrating advanced control flow attacks as well as DOP attacks through heap-resident variables. To this end, we explored the use of multi-variant execution (MVX) with each variant having uniquely seeded random heap allocators. This capability enables our system to automatically track heap allocation pointers without the need for storing explicit meta-data. We then re-randomize heap allocations to thwart attacks that perform runtime probes to discover allocations. This technique will provide modular heap allocation protection while maintaining compatibility with legacy binaries.

In all, this thesis presents novel techniques that carve out a new space in advanced code-reuse attack protections, offering a protection strength as good or better than prior solutions. These techniques provide additional protections for advanced control flow attacks and DOP attacks, while incurring minimal performance overheads.

Chapter 1
Introduction

Society's reliance on computing platforms has continued to rapidly increase over the past few decades. Computing devices play a vital role in our daily lives: these devices keep private information (e.g., bank accounts, passwords, etc.) as well as keep track of our activities (e.g., browsing history, location information, etc.). The bulk of information gathered and accessed via these devices has made them an attractive target for malicious actors to perform attacks in order to gain access to this information. This is exacerbated by the fact that many of these devices are interconnected through internet, enabling remote attacks.

1.1 A Brief History of Exploits and Defenses

Memory corruption is the leading attack vector used to compromise systems security. Attackers perform this attack by exploiting memory bugs in the program, such as buffer overflow, dangling pointers and format string vulnerabilities to corrupt sensitive data, like code pointers. The root cause of these vulnerabilities is that performance bound applications, including most legacy systems, are written in type-unsafe languages, such as C and C++, which are inherently prone to memory errors. Performance bound applications include most low level system code, spanning from the operating system kernels to language runtime of type-safe languages. Memory corruption attacks typically target *control-plane memory*, which consists of data used in control-flow transfer instructions, such as return addresses and function pointers. These attacks are commonly known as control-flow hijacking attacks, in which an attacker sends a maliciously crafted message to the target machine in order to eventually control the execution flow of the program.

In the early days, memory errors were exploited in stack smashing attacks, in which an attacker overflows a stack resident buffer to inject shellcode and clobber adjacent control data, such as a return address, in order to redirect the execution flow of a vulnerable application to the injected shellcode [1]. To mitigate stack smashing, stack canaries [2] are

introduced where a random value is inserted to precede the return address on the stack, and the integrity of the canary is checked before the function returns. This protection has led attackers to overflow a heap resident buffer to overwrite heap resident control data instead [3][4]. In response to these attacks, several mitigation techniques have been introduced, the most widely adopted being Address Space Layout Randomization (ASLR) [5] and Data Execution Prevention (DEP) [6][7]. ASLR randomizes the base address of data sections, as most attacks rely on discovering the absolute address of the code to jump to. An early attack to circumvent ASLR was heap spraying, where an attacker sprays a large portion of the program's heap with shell code (which was achievable in 32-bit systems) to increase the chances of successful exploitation. DEP, in contrast, marks all writable pages non-executable (read/write-only), rendering all code injection attacks ineffective.

To circumvent DEP, attackers began to reuse existing code in the form of return-into-libc [8] and its more general form, return-oriented programming (ROP) [9]. These attacks utilize an existing library function (*i.e.*, return-into-libc) or chain of instruction sequences that end with an indirect jump (*i.e.*, ROP), commonly known as code gadgets, to implement an arbitrary operation. Various mitigation techniques have been proposed to defend against these attacks, the most prominent one being control-flow integrity (CFI) [10]. CFI is a principled approach that restricts all indirect control flows to adhere to a statically determined control-flow graph (CFG).

1.2 Advanced Attacks

CFG Mimicry Attacks CFI inserts checks before indirect branches to ensure that all indirect control transfers are within the CFG. The effectiveness of CFI is strongly tied to the precision of the control-flow graph it enforces. Based on the precision of their CFG, CFI techniques can be broadly categorized as coarse-grained and fine-grained. Coarse-grained CFI techniques, such as CCFIR [11], relax the CFG to achieve practical solutions. CCFIR categorizes target addresses into only three sets: return, indirect call and indirect jump. For example, CCFIR only enforces returns to target a call-preceded instruction. Because of their low overhead, coarse-grained CFI techniques have gained wide spread adoption in industry – Microsoft Control Flow Guard and Intel Control-flow Enforcement Technology are such examples. However, recent advanced attacks generally referred to as *CFG mimicry attacks*, like control flow bending [12] and counterfeit object-oriented programming [13], have shown that control-flow attacks are still possible without leaving the CFG. These attacks piggyback on the relaxed CFG by swapping targets of indirect control-flow instructions with

another one from the allowed set of targets to circumvent these CFI solutions. Recently proposed fine-grained CFI solutions, such as cryptographically enforced CFI (CCFI) [14], have been shown to be immune to these attacks. CCFI is a compiler transformation that uses a cryptographically-secure hash-based message authentication code to enforce the CFG. However, its high performance overhead (52% for SPEC'06) deemed it impractical for production environments.

DOP Attacks In contrast to control-flow attacks, non-control data attacks manipulate the data plane memory of the program or data that is not directly used for control-flow transfer. These attacks do not need to directly modify the control-flow of the program, and thus cannot be detected by traditional control-flow protections. Instead, these attacks corrupt sensitive data used for decision making in the program to escalate privileges [15], leak secret keys (HeartBleed) [16] and import untrusted code into browsers [17]. Early measures against non-control data attacks provide protection of security critical data (kernel data [18], programmer annotated critical data [19]). However, a recent advanced technique called data-oriented programming (DOP) [20] has generalized non-control data attacks to achieve Turing-complete computation by repeatedly corrupting any non-control data in order to execute a sequence of instructions embedded in the legitimate control-flow.

Figure 1.1 shows an overview of how memory corruption attacks morphed over time to these advanced code-reuse attacks. In this dissertation, we make the following key observations for stepping up protections to mitigate these advanced attacks:

- Keeping the program on the CFG is important for reducing the control-flow attack surface. However, the effectiveness of CFI protections is highly correlated with the precision of the enforced CFG. To mitigate CFG mimicry attacks in production systems, it is essential to have a CFI solution with a more precise CFG and minimal performance overhead.

- Even though complete memory safety enforcement techniques can stop advanced code-reuse attacks, they are impractical because of their very high overhead. Lightweight enforcement techniques, such as stack canary, are ineffective in the face of memory disclosure (*e.g.*, Just-in-time ROP [21]) and systematic brute force attack (*e.g.*, Blind ROP [22]). A more durable and practical approach is adding uncertainty in the layout of the program using randomization and periodic re-randomization (*churn*) to mitigate runtime disclosure of the layout.

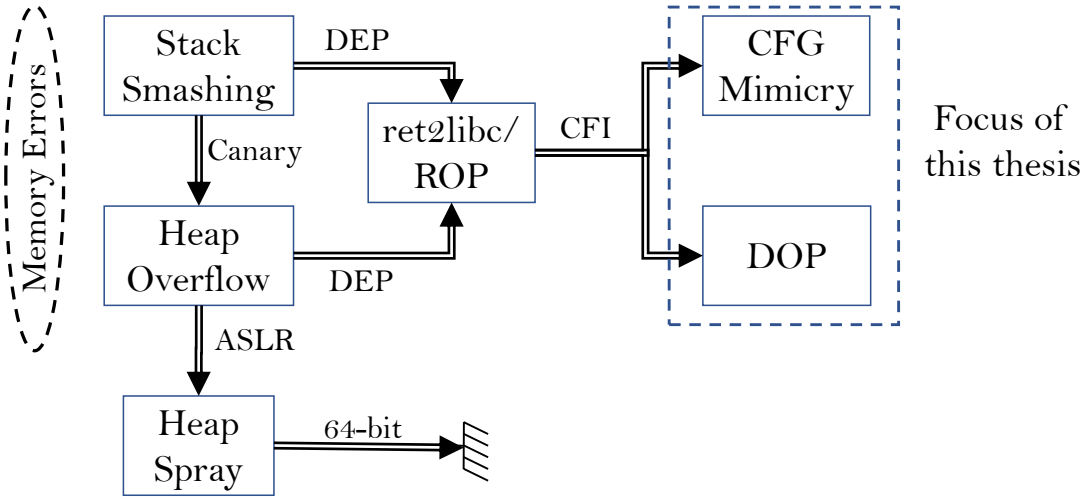


Figure 1.1 Overview of the evolution of code-reuse attacks show how the memory corruption exploit landscape evolved through time.

1.3 Contributions

In this dissertation, we present techniques to mitigate against advanced code-reuse attacks that do not leave the CFG of the program. First, we explore a novel approach to protecting CFG called ProxyCFI. Instead of building protections to stop code pointer abuse, ProxyCFI replaces code pointers whole-sale in the program with a less powerful construct: pointer proxies. Pointer proxies are not pointers; rather, they are random identifiers associated with legitimate control-flow edges. Pointer proxy values are defined per-function and are randomized at program load-time to mitigate advanced CFG attacks. If a program wants to make an indirect jump to a code entry point, it must express this jump with the correct pointer proxy; otherwise, the program is terminated. To ensure that the approach covers all control-flow (thus, the attacker cannot leave the control-flow graph and introduce unchecked indirect jumps), our implementation includes a load-time verifier that performs reachability analysis of code to ensure that there is no vulnerable control and a loader that marks all code pages unreadable to mitigate runtime discovery of the randomized pointer proxies. ProxyCFI delivered these protections with only a 4% average slowdown, stopping a broad range of real-world attacks and 100% of the RIPE x86-64 attack suite, while accommodating existing programming frameworks, including shared libraries.

Second, we evaluated the effectiveness of prior stack layout randomization techniques and show that they are ineffective at stopping real-world DOP exploits. We then propose Smokestack, a runtime stack-layout randomization technique that addresses the problems with previous approaches. Smokestack instruments programs to randomize their stack layout at runtime for each call. By doing so, Smokestack minimizes the utility of information

Table 1.1 Summary of proposed defenses The unshaded circles show no protection, half shaded circles show partial protection, and fully shaded circles show full protection.

	CFG	Stack	Heap
ProxyCFI	●	◒	◒
Smokestack	◒	●	○
Heap Re-randomization	◒	○	●

gained in the probes of a chained DOP attack for later attack stages. We evaluated our runtime stack layout randomization technique on both CPU intensive and I/O bound applications as well as investigated its security, performance and memory usage. Our results demonstrate that Smokestack, with a cryptographically secure pseudo random generator, is effective in stopping DOP attacks with an average slowdown of 8.7%.

Finally, we present HeapRand, a runtime technique to randomize heap allocations to prevent attackers from orchestrating advanced control flow attacks as well as DOP attacks through heap-resident variables. HeapRand uses multiple replicas of a program with each having uniquely seeded random heap allocator. This will allow the system to automatically track heap resident pointers without the need for storage meta-data. We use these to re-randomize heap allocations to thwart attacks that perform runtime probes to discover allocations to orchestrate advanced code-reuse attacks. HeapRand provides a modular heap allocation protection and is compatible with legacy binaries.

In all, this dissertation proposes novel and practical techniques in the landscape of advanced code-reuse attack protections, offering security guarantees as good as or better than prior proposed techniques and providing defenses for CFG mimicry attacks and DOP attacks, but with significantly lower overheads. Table 1.1 summarizes the proposed defenses.

1.4 Dissertation Road map

The remainder of this dissertation is organized as follows. Chapter 2 presents a detailed background on advances in code-reuse attacks and illustrates concepts that are vital to understand the remainder of this dissertation. Chapter 3 details how the use of load-time randomized per-function pointer proxies loaded to an execute-only code region can deter CFG mimicry attacks. In Chapter 4, a stack-layout randomization technique that mitigates DOP attacks is discussed. In Chapter 5, we present `HeapRand`, a heap layout randomization technique using multi-variant execution. Lastly, Chapter 6 concludes the dissertation and hint towards future directions in advanced code-reuse defenses.

Chapter 2
Background

In this chapter, we present a detailed background on memory corruption exploits and their development over time. We start by describing the major categories of memory corruption exploits based on the type of memory they target. Then we show how the attack surface has persisted through time, adapting to various defenses that have been proposed and deployed on modern systems. Finally, we show the recent advances in code-reuse attack that this dissertation addresses.

2.1 Memory Errors

Programs written in low level languages are vulnerable to memory errors. In most cases, memory errors force the program to show an erratic behavior or crash. These errors can also be exploited by an adversary to perform malicious operations on the victim system. These operations can range from crashing the program repeatedly for mounting a Denial of Service attack (DoS) [23] to executing unintended operation in the program for extracting sensitive information. In general, memory errors constitute two major categories: spatial and temporal memory errors.

Spatial Memory Error Spatial memory errors enable a memory access to go beyond the bounds of an allocated object. A classic example of spatial memory errors is buffer overflow, during which a buffer is overwritten beyond its allocation bounds to read or modify the content of an adjacent memory locations. Buffer overflows can happen, for example, when performing a memory copy operation if the destination buffer is not big enough to hold the content of the source buffer. This typically happens if the source buffer is supplied by an adversary and the destination is a fixed size buffer. Buffer overflow bugs can be exploited to corrupt sensitive control-flow data, such as a return address or function pointers, to hijack control-flow of a program.

Temporal Memory Error Temporal memory error happens when a program dereferences a dangling pointer. With temporal memory errors, a dangling pointer is dereferenced after the memory allocation it points to has been freed. Programs typically have multiple pointers to a single allocation, as codes are usually shared across different parts of the program. Dangling pointers occur when the object is deallocated with one of the references to the allocation and the program keeps using the other references to the allocation afterwards. In the mean time, another allocation may occupy the location that has been freed. Hence, successive dereferencing of the dangling pointers of the previous object will erroneously access the new allocation to corrupt other pointers or data inside the new allocation. Most temporal errors target heap allocations, but it is also possible to have dangling stack pointers when a stack pointer is assigned to global scope pointers, which can be exploited to corrupt sensitive stack resident data, such as a return address to hijack control-flow.

Based on the type of memory they exploit, memory corruption attacks can be categorized into two major categories: control flow attacks and non-control data attacks.

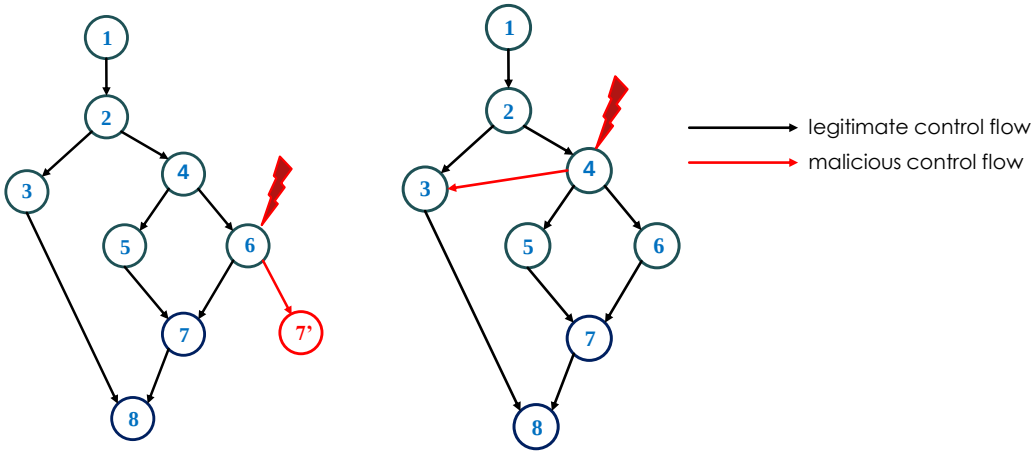
2.2 Control Flow Attacks

Control flow hijacking attack is the primary means of exploitation for memory corruption vulnerabilities, as it enables an adversary to execute an arbitrary code in the victim system. The early variants of control flow hijacking attacks involve code injection, in which an adversary uses buffer overflow vulnerability in the program to inject malicious payload in the address space of the program and then redirects control flow to the injected malicious payload by corrupting a control flow data, such as return address. In response to a number of principled defenses introduced in modern systems, recent advances in exploit involve code-reuse attacks, such as return-oriented programming (ROP) [9] and its variants, have surfaced [24][25].

Code Injection Attacks Figure 2.1a illustrates code-injection attacks. In these attacks, an attacker introduces a new malicious node in the control-flow graph of the program during the program runtime. In the example, the node 7' is introduced in the normal control flow of the program. This is typically accomplished using a memory corruption vulnerability in the program, such as buffer overflow, to add a sequence of malicious instructions in the address space of the program. This attack is complete when an attacker manages to divert the control-flow of the program to execute the injected malicious payload. This can be achieved by corrupting a control-flow data to point to the injected code, instead of the

legitimate control flow destination.

Code-reuse Attacks Code-reuse attacks are typically used on systems hardened with data execution prevention capabilities. The principle behind code reuse attacks is to redirect the execution flow of the program to unintended instructions that are already in the address space of the program to perform malicious operations. There are various ways that code-reuse can be achieved. Early attacks in this domain involve an adversary redirecting the program execution to an existing library function (*e.g.*, `system()` function in libc) to perform an attack [8]. The more general approach for this attack vector is return-oriented programming (ROP) [9]. ROP stitches together short sequences of instructions in the address space of the program, known as gadgets, to allow an adversary to perform arbitrary Turing complete computation in the victim system. Gadgets are chosen to end in an indirect control-flow instruction in order to chain them together.



(a) Code injection attacks

(b) Code-reuse attacks

Figure 2.1 Control flow hijacking attacks

2.3 Non-control Data Attacks

With the widespread adoption of control-flow hijacking attack defenses, attacks through corrupting non-control data have become prevalent. These attacks allow an adversary to access data that is not directly used in control flow transfer from the address space of the program. These attacks are typically used to tamper with a security critical program data (*e.g.*, to achieve a privilege escalation) or leak sensitive information. For example, overwriting the `Safemode` flag on Internet Explorer 10 has been shown to achieve privilege escalation attacks capable of running arbitrary code in victim process[15]. Another infamous example

of non-control data attacks is the Heartbleed [16] attack in OpenSSL, in which the the bug in the program enables a buffer over-read of adjacent memory to leak sensitive information, such as secret keys.

2.4 Memory Corruption Defenses

There is an enormous body of work in memory corruption defenses. Various memory corruption mitigation techniques have been proposed to upend attacks at different stages of the exploit. These proposed solutions can be broadly categorized as enforcement based techniques and randomization based techniques.

2.4.1 Enforcement Based Defenses

These defense techniques generally work to enforce lower level policies in order to deter memory corruption attacks at various stages. Some techniques enforce policies to stop the problem at the source while other techniques involve stopping the exploitation of the vulnerability. Complete memory safety is a typical example of the former while control-flow integrity is an example of the later.

Memory Safety

High-level languages achieve memory safety with builtin runtime bounds checking to protect against spatial memory errors and garbage collection, ultimately protecting against temporal memory errors. Low-level languages, such as C and C++, trade these features for performance and hence do not provide any memory safety guarantees. Memory safety techniques try to regain these features by instrumenting the program so that all memory accesses stay within the bounds of the allocated objects in order to completely eliminate memory corruption attacks from their source. Most proposed memory safety defenses are either pointer based or object based.

Pointer based techniques work by storing a lower and upper bound metadata for a pointer and adding checks at runtime, ensuring the memory accesses through the pointer are within the bound. Fat pointers [26] extend the pointer representation to a structure which also stores the bounds metadata alongside the data. Changing the representation of pointers, however, alters the memory layout, resulting in binary incompatiblity. Softbound [27] addresses this issue by storing the metadata in a shadow memory region. Another alternative approach

is low-fat pointers [28], in which the bounds metadata is implicitly encoded in the address of pointers. This approach requires customizing stack and heap allocators to associate allocation sizes with finite sets of addresses.

Object based approaches, on the other hand, work to detect out-of-bound memory accesses by associating the bounds metadata with the allocated objects, focusing on pointer arithmetic operations instead of dereferces of pointers. Unlike pointer based approaches, object based approaches do not allow pointers to go beyond the bounds even if they are not referenced. Address Sanitizer (ASan) [29] is a popular memory error detector capable of detecting out-of-bounds memory accesses of allocations in various sections of memory, including global, stack and heap. ASan stores metadata for each allocated object in a disjoint memory region to detect erroneous memory accesses based on a corrupted pointer. Baggy Bounds Checking [30] is the state-of-the-art in this domain. Baggy bounds checking trades memory overhead for performance by adding padding to allocated objects in order to align the base address to be a multiple of their size, making their size to be the nearest higher power of two. This affords Baggy Bounds Checking an effective bounds lookup operation in addition to the compact way of representing the bounds meta-data.

Even though complete memory safety techniques are the most effective means to stop memory corruption attacks, their performance overhead makes them unfavorable for production systems. For example, Softbound provides a spatial memory safety, incurring an average overhead of 116% , and CETS [31] provides temporal memory safety, incurring an average of 48% on SPEC CPU benchmarks. Due to their high overhead, complete memory techniques are more suited for debugging purposes.

2.4.2 Control-flow Integrity

A typical program is composed of a number of subroutines interacting with each other to perform a certain task. Each subroutine performs a certain task and transfers control to the next subroutine. All the possible paths by which the subroutines of a program interact with each other defines the control-flow graph (CFG) of the program. CFG of the program is composed of blocks of instruction that are interconnected with edges, showing all the possible routes the program execution can take. CFG edges can be used for forward edge control-flow transfers and backward edge control-flow transfers. Forward edges are control-flow transfers that direct execution flow to a new code location and are typically used to represent indirect jump and indirect call instructions. Backward edges, in contrast, are used to return to an earlier code location that was involved either in a direct control-flow transfer, such as direct calls, or indirect forward control-flow edges. CFG of a program can be discovered through

source-code analysis, binary analysis, or execution profiling. Malicious control-flow transfers due to attacks, such as code injection return-oriented programming, result in violation of the CFG of the program, as they typically use a new edge that is not present in the benign CFG specified by the programmer. Even though data execution prevention is capable of stopping code injection attacks, it can be circumvented by code-reuse attacks.

Control-flow Integrity (CFI), originally introduced by Abadi et al. [32], enforces a policy to restrict the set of potential targets of an indirect control-flow transfer. Even though it does not mitigate the initial memory corruption, CFI ensures the validity of code pointers before they are used. CFI works by determining the valid set of targets of all indirect control flow instructions, including calls and returns, to construct the CFG of the program. Before each indirect control flow, a check is performed to ensure the target address is within the valid set of targets for instruction during the runtime of the program. If it is not within the valid set, a violation is issued. This puts a strict limit on the choices an attacker has to overwrite a code pointer.

```
1  bool lesser(int x, int y){ return x < y;}
2  bool greater(int x, int y){ return x > y;}
3
4  bool sort(int a[], comp fptr){
5    ...
6    if (fptr(a[i], a[i+1]))
7        swap(a[i], a[i+1]);
8    ...
9  }
10 void sort_asc(int a[]){
11     sort(a, lesser);
12 }
13
14 void sort_dsc(int a[]){
15     sort(a, greater);
16 }
```

Listing 2.1 Example control flow in a program.

Figure 2.2 illustrates the instrumentation introduced by CFI for the example program in listing 2.1. CFI assigns a unique identifier for each possible target of an indirect control-flow transfer and then gathers a set of all valid identifiers for each control-flow transfer instruction in the program. These sets are used in the checks, which are inserted before an indirect control-flow instruction is executed to ensure that the instruction only targets valid identifiers in the target set. For instance in figure 2.2 the return instruction of `sort` function can only target valid targets in set `L2`, which contains addresses `a1` and `a2`, the only addresses that

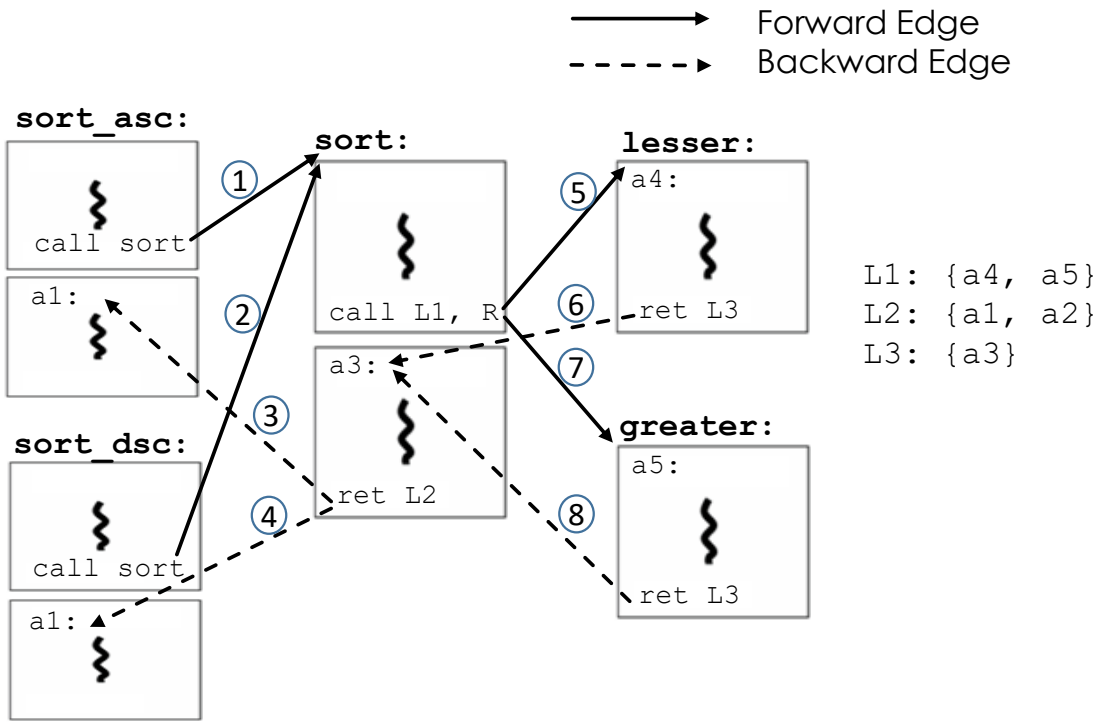


Figure 2.2 Control-flow transfer checks inserted by CFI

are next to calls to function `sort`. If the target of the return instruction is not within the valid set, a violation is issued.

Several other enforcement techniques have been proposed in the past, including checking the integrity of code pointers, data flow graphs and points-to sets. Code-pointer Integrity (CPI) [33] provides more comprehensive memory safety for code-pointers by isolating them, including their bounds information from other memory regions. Data-Flow Integrity (DFI) [34] performs checks to detect the corruption of data before it is accessed by read instructions. DFI enforces an integrity check of reads based on the instruction that wrote to the memory location that is being read. Write Integrity Testing (WIT) [35], on the other hand, restricts pointer dereferences for write accesses to accomodate only objects in their points-to set gathered through pointer analysis of the program.

2.4.3 Randomization Based Defenses

Randomization based defenses aim to obfuscate targets of memory corruption attacks by randomizing various assets used in the program, including location of program segments, layout of the code section, layout of the data segment or even the data itself in order to make

memory corruption attacks result in unpredictable behavior. These defenses are effective in stopping attacks as attackers typically rely on the certain details of the program to perform a successful exploit. For example, control flow hijacking attacks rely on finding the location of control data to be overwritten in order to divert the execution flow of the program. Eliminating the predictability of certain assets in the system can effectively limit the success of attacks that rely on them. To circumvent this, attacks have to either rely only on assets that are not diversified or adapt to account for the diversity introduced by the defenses.

Earlier efforts in randomization based defenses include instruction set randomization [36] that works to mitigate code injection and tampering with existing code. However, the wide spread deployment of data execution prevention (DEP) and read-only code pages has outdated this defense.

The other randomization based defense approach involves diversifying the layout of various sections of the program. The most widely deployed randomization based defense in this domain is Address Space Randomization (ASLR). ASLR randomizes the locations of different memory segments in the address space of the program, including stack, heap and code segments. Certain implementations of ASLR suffer from some weaknesses, including low entropy and not randomizing code regions. For example, However, the biggest threat for ASLR is memory disclosure, as a single information leak is capable of de-randomizing it. There are numerous fine-grained layout randomization based techniques that have been proposed to close the shortcomings of ASLR, including randomization of code section at function level [37], basic block level [38][39] and instruction level [40][41]. Yet another approach for randomization based defenses is data space randomization. These defenses randomize the representation of data as it resides in memory. Pointguard [42] encrypts data while it is stored in memory and decrypts it before it is used in operations. However, Pointguard uses a single key to XOR all pointers for encryption and decryption. So, if a single known encrypted pointer is leaked, the key can be recovered to de-randomize the whole memory. Data Space Randomization [43] addresses the issues of Pointguard by using a different key for all variable, including all pointers in the program.

Another important aspect in randomization based defenses is the frequency of randomization. Randomization can be performed at various stages of the program life cycle, including during compile time, load time and run time of the program. This has an important implication for the effectiveness of these defenses. ASLR, for example, randomizes sections of the program at program load time. So, if the randomization offset is discovered during runtime of the program, then it can be bypassed to mount a successful attack. Runtime randomization techniques are immune to de-randomizing efforts; they mutate various aspects of the program during runtime. A recent example of runtime randomization technique is Morpheus

[44], which periodically changes the location of pointers using a random displacement for code pointers and data pointers. Morpheus also periodically re-randomizes the encryption keys used to encrypt code and pointers while they are in memory.

2.5 Advanced Code-reuse Attacks

In this section, we detail advanced code-reuse attacks that are capable of bypassing control flow integrity (CFI) based defenses. These attacks can be grouped in to two major groups. The first category includes attacks that still manipulate control data while staying in CFG of the program. We call these attacks CFG mimicry attacks. The other category is data-oriented attacks, which only manipulate non-control data and hence do not violate the control-flow integrity of the program.

2.5.1 CFG Mimicry Attacks

Classic code reuse attacks, such as return-oriented programming, enable an adversary to execute arbitrary computation on the victim system by reusing existing code. Control-flow integrity protects against these attacks by enforcing the execution flow of the program to follow the programmer specified control flow graph. CFI techniques typically achieve these protections by instrumenting the program in order to check if the destination of an indirect control transfer at runtime is within the allowed set of targets. Hence, CFI techniques still allow modification of control-flow data, such as return addresses and indirect branch targets, as long as the new indirect branch target is within the allowed set of targets in the enforced control-flow graph. A typical example is Control-flow bending [12]. These attacks involve finding useful gadgets in the legitimate control flow of the program. Then these attacks exploit a memory corruption vulnerability to corrupt code pointers within the allowed set to perform malicious computation while piggybacking on the benign CFG of the program.

For example, in Figure 2.2, let as assume a call to `sort_asc` is followed by a call to `sort_dsc`. CFI inserts a check before the return `sort` functions to ensure that it only takes either edge 3 or edge 4. Under normal execution, a call to `sort` through edge 1 is followed by a return through edge 3 and a call through edge 2 is followed by a return through edge 4, even tough both targets are valid at any point according to the enforced CFG. CFG mimicry attacks manipulate this by allowing the first call to `sort` from `sort_asc` to return normally through edge 3, but these attacks manipulate the call from `text_dsc` to return through edge 3 instead of its normal return path, edge 4. This affords the attacker

to execute gadgets available in between the two call sites as many times as needed. This technique is known as loop injection and is used to achieve Turing complete computation without violating the control-flow integrity of the program.

2.5.2 Data-Oriented Attacks

Data-oriented attacks can take various forms based on how an adversary manipulates non-control data resident in the address space of the program. Early attacks in this domain involve direct data manipulation, wherein an attacker directly manipulates a specific target data in the program memory to accomplish the malicious purposes. To perform a direct data manipulation attack, an attacker needs to know the exact location of the target data in the address space of the program. Chen et al. [15] demonstrate the serious implication of non-control data by showing how they can be used to corrupt or leak security-critical data.

Data-Orientated Programming is a generalization of data-oriented attacks to perform expressive computation without relying on tampering with code pointers. DOP attacks instead tamper with non-control-data in order to induce the execution of sequences of instruction within the normal program flow, using an attacker controlled input. The attack consists of a data-oriented gadget performed by each sequence on top of normal program logic. DOP gadgets are stitched together using a gadget dispatcher, which could be a loop whose counter is controlled by the attacker in order to chain together data-oriented gadgets within the loop to perform Turing-complete computation. Listing 2.2 shows a program vulnerable to DOP attacks. If the input bounds of function (`get_input()`) are unchecked, an attacker can control variables `size`, `step`, `ctr` and `req`.

```
1  func() {
2   int *ctr, *size = 0, *step = 1;
3   char *buff[LEN]; int *req;
4   for(;ctr < MAX; ctr++){
5     get_input(buff, req); //vulnerable function
6     if(*req == 0)
7       *size += *inc;
8     else if(*req == 1)
9       *size -= *inc;
10    else
11      *step = *req;
12  }
13 }
```

Listing 2.2 Example DOP attack.

DOP attack grants an adversary the ability to perform addition, subtraction and copy operations on any memory value, in any order desired by the attacker. Hu et al. [20] demonstrate a DOP attack on ProFTPD capable of bypassing randomization based defenses (such as ASLR) in order to leak the private key of the OpenSSL server.

2.5.3 Key Characteristics

Advanced code-reuse attacks can be characterized by the requirements that they introduce to adapt to deployed defenses. These requirements have been growing due to prevalence of various defenses built in modern systems, including DEP, ASLR and stack canaries. For example, DEP has added finding an address of a gadget using static analysis or memory disclosure as a requirement to perform a code-reuse attack. The most common characteristics required by advanced code-reuse attacks include memory disclosure, non-linear overwrite, relative address attacks and partial overwrite vulnerabilities.

Memory Disclosure ASLR has been deployed on virtually every modern system due to its insignificant performance overhead. Consequently, bypassing ASLR has become a precursor for any successful memory corruption exploit. On 32-bit implementations of ASLR brute force attacks were viable because of its low entropy. On 64-bit systems an attacker would require more than that to bypass ASLR. The most common technique used to bypass ASLR on 64-bit systems is memory disclosure, by which an adversary leaks a runtime address. A single memory address leakage is sufficient to randomize ASLR, as it only randomizes the base address of sections and hence the offsets with the section always remain the same. Memory disclosure attacks are generally used to circumvent randomization based defenses for de-randomizing layouts or extracting secret keys used for the randomization.

Non-linear Buffer Overflow Stack canaries [2] are the first line of defense proposed to protect against stack smashing attacks. This defense works by inserting a secret value between return address and the rest of the stack allocations, which are checked before returning to detect whether the return address is corrupted by buffer overflow. Similarly, ASAN [29] adds redzones around objects in the stack, heap and global sections to detect spatial memory bugs when the redzone is corrupted. Non-linear overflow bypasses these type of defenses, as it enables an attacker to overwrite an arbitrary address without corrupting adjacent locations. This could be achieved, for example, when an attacker has control over the index of an array that is accessed without any bounds checking, where in a user supplied index of the array is

made to point to an arbitrary address in the program. This is commonly caused by integer overflow related memory errors.

Relative distance based attacks require only relative offset between allocations; for example, in non-control data attacks, an attacker only needs to know the relative distance between a buffer that has a buffer overflow vulnerability and the location of the security critical data to be corrupted. These attacks are generally immune to base address randomization protections, such as ASLR, since the relative offset stays the same.

Chapter 3
Wrangling in the Power of Code Pointers

3.1 The Unending Cycle of Control-flow Attacks

For more than four decades, control flow attacks, in which attackers force programs into executing code sequences unanticipated by the developer, have played an important role in the infiltration of secure systems. These attacks are particularly attractive to attackers because they provide the immediate agency necessary to deploy attack payloads, leak important information, embed a rootkit, launch an additional attack (such as, privilege escalation), etc. As such, there has been much attention paid to reducing systems' vulnerability to control flow attacks.

Early measures to stop control flow attacks included the StackGuard [45], which deterred the overwriting of return addresses by placing a random canary word next to it and checking the integrity of the canary before jumping to the address pointed by it. Shortly after the use of canaries became prevalent, the injected code moved into the heap, using attacks such as double-free attacks [46], heap overflows [3] or heap spray attacks [4]. Code injection into the heap resulted in the locking down of the entire data segment from code execution (*e.g.*, data execution prevention [6] [7]), which in turn led to return-oriented programming (ROP) techniques by which existing code sequences ending with a return are reused to create new attacker-selected code sequences [9]. ROP attacks have motivated the wide use of address space layout randomization (ASLR) [5], by which the location of code is changed each time a program is executed. However, pointer leaks [47] and brute-force attacks [48] [49] have skirted even this powerful defense. This led to the emergence of more advanced defense techniques, the most prominent one being Control-flow Integrity (CFI).

CFI [10] follows a principled approach to mitigating control flow attacks by enforcing the runtime execution path of a program to adhere to the statically determined CFG. It takes this approach by checking if the target of an indirect jump is within a valid set of targets. However, prior proposed CFI solutions are either impractical or ineffective. Some, which strictly follow a program's CFG, have high overheads that render them impractical

[14]. Others attempt to reduce overheads by approximating the CFG with limited classes of targets (*e.g.*, two classes for function pointers and return addresses) [11][50][51][52], but these do not protect against control flow attacks that swap targets while remaining on the CFG [53][12][54][13].

In this work, we make the key observation that many of the vulnerabilities in control flow stem from the excessive power inherent in code pointers. To stem the tide of control flow attacks, we propose a novel approach to control flow integrity, called ProxyCFI [55], that replaces all code pointers in the program with *pointer proxies*. A pointer proxy is a unique random identifier (64-bits in our implementation), which represents a forward or backward control flow edge from specific exit point to specific entry point in the program. Wherever in the program a code pointer lies, it is replaced with its corresponding pointer proxy. Consequently, all indirect jumps in the program (*e.g.*, returns and jumps-through-register) are replaced with a multi-way branch that implements a direct jump to the address associated with the pointer proxy. As pointer proxies are a function of both the source and the target of an edge, swapping pointer proxies results in a violation even if they have the same target.

Using this approach for indirect control flow, ProxyCFI provides a strong form of CFI, so that the program gains a number of highly desirable properties. First, by replacing all indirect jumps with fully enumerated multi-way branches, the program code is fully discoverable at load time using only a single breadth-first traversal of the program's CFG. Second, if the program replaces *all* code pointers with pointer proxies and *all* indirect jumps with multi-way fully enumerated direct branches, it becomes impossible for control flow to escape the control flow graph implemented by the program. In fact, there is simply no direct path in these programs from the data segment to the program counter; thus, all PC updates are via direct branches specified by the programmer.

To ensure that all executions stay on the program CFG for even third-party-generated ProxyCFI compliant code, a binary-level program verifier first validates that programs and libraries have CFGs that are fully discoverable, use only pointer proxies, and avoid all indirect jumps/returns. Once validated, execution cannot leave the programmer-specified CFG. Finally, to thwart a highly motivated attacker who studies the code in advance to discover the code entry points associated with pointer proxy values, the verifier assigns a load-time-unique set of random pointer proxies before the program begins execution. In addition, the verifying-loader marks code sections unreadable, to protect from active-read attacks that gather pointer proxies using memory leaks.

More importantly, ProxyCFI has a number of powerful features to deter attacks that mimic legitimate control flow (*i.e.*, control flow attacks that seemingly remain on legitimate

Table 3.1 Comparison of Code Pointers to Pointer Proxies. Pointer proxies preserve program control integrity by reducing their capabilities. This table lists the differences in capabilities between code pointers and pointer proxies. Ultimately, it is the powerful nature of code pointers that enable many CFG attacks.

	Code Pointers	**Pointer Proxies**
Arithmetic Allowed	Yes	No
Totally Ordered	Yes	No
Trivial Forgery Attacks	Yes	No
Permit Relative Distance Attacks	Yes	No
Replay Attacks on Returns and fptrs	Yes	Only from the same source address

control flow edges), such as control flow bending (CFB) [12]. These attacks exploit the fact that existing CFI techniques allow executions to maliciously divert indirect branches if the target address is still in the valid set of targets. ProxyCFI thwarts this attack, as a pointer proxy is unique to a particular source and target address, which makes a pointer proxy used in one function context invalid in another even if they share the same target addresses.

Table 3.1 lists the comparative capabilities of traditional code pointers versus pointer proxies. As shown in the table, pointer proxies do not support arithmetic manipulation; thus, relative-address based control flow attacks, such as ASLR derandomization attacks [49], would not be possible with pointer proxies. Moreover, pointer proxies are much more difficult to forge, since their assignment is not in anyway related to other pointer proxies, whereas pointer values often reveal much information through relative address distances to other code objects, facilitating relative address inspired attacks. Since pointer proxies are unique to a given function, return address copy attacks, such as the return-into-libc [8] and backward-edge active-set attacks [56], become more challenging, as the pointer proxies of other functions (which are assigned at load-time) must be leaked and then translated to the local function's proxies (which have no correlation even if the current function calls the intended target).

3.1.1 Contributions of This Work

In this thesis, we introduce ProxyCFI, a novel control flow integrity technology that works to deter advanced control flow attacks while incurring lower performance overhead. Instead of putting protections in place to enforce proper indirect jumps and returns, our

approach replaces all code pointers in the program with pointer proxies, which lack many features afforded to traditional code pointers. Specifically, this thesis makes the following contributions:

We present a novel control flow enforcement technology dubbed ProxyCFI that provides an efficient and practical enforcement of the programmer-specified control flow, while also providing protections against advanced CFG reuse attacks. This enforcement is achieved with a relaxed threat model that assumes the attacker has read and write control over all data memory.

Then, we detail the end-to-end implementation of ProxyCFI within the GNU GCC compiler toolchain. Our implementation includes support for shared libraries and a verifying loader that allows the safe introduction of third-party codes without fear of compromising control flow integrity. In addition, our loader further protects control flow by assigning program pointer proxies at load time and marking code pages unreadable.

Moreover, We demonstrate the efficiency of the approach by running a wide range of CPU-centric and network-facing applications. In addition, we implement two pointer proxy compile-time optimizations: profile-guided sled sorting and function cloning, which ultimately reduce the slowdown of this technology to only 4% on average. In addition, our security analyses shows that the technology stops real-world control flow attacks, including attacks that mimic legitimate control flow, and also demonstrating 100% coverage for the RIPE x86-64 control flow attack suite [57].

The remainder of this chapter is organized as follows. Section 3.2 details the ProxyCFI technology. Section 3.3 details our implementation of ProxyCFI in the GNU GCC C/C++ toolchain. Section 3.4 presents a detailed performance and security analyses of the ProxyCFI. Section 3.5 compares ProxyCFI to a wide array of previous control flow integrity techniques, and Section 3.6 concludes the chapter.

3.2 Protecting Control Flow with ProxyCFI

In this section, we detail our threat model and the broad ProxyCFI concept, and then present how to build and verify programs (including shared libraries) with pointer proxies.

3.2.1 Threat Model

In this work, we assume a very powerful attacker who wants to redirect control flow to a code sequence that deviates from the programmer-specified CFG. In accomplishing their

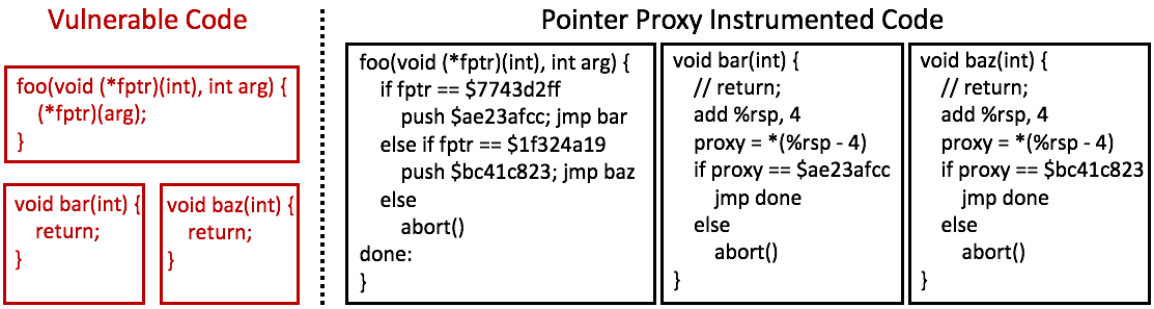


Figure 3.1 Example Code Sequence using Pointer Proxies. Pointer proxies replace the code pointers in a program with per-function random identifiers associated with legal code targets. Multi-way direct branch sleds translate pointer proxies into direct jumps. As such, pointer proxy programs lack the use of indirect jumps (*e.g.*, jump-through-register or returns), and thus, it is not possible to leave the programmer-specified control flow graph.

control flow attack, the attacker has read and write access to any data location, including globals, stack and heap variables, as well as data storage locations holding pointer proxies. The code segment of the program is assumed to be non-writable.

The programmer-specified CFG includes the basic blocks of the program connected by edges specified by the direct jumps in the program. For indirect calls, the programmer-specified CFG is assumed to allow the program to jump to any same-typed function (as the function pointer) entry point which has had its address taken (&) somewhere in the program. Finally, the indirect jumps made by returns are assumed to jump to any instruction immediately following a legal calls (direct or indirect) to the returning function.

Given this powerful attacker, ProxyCFI work to prevent the attacker from hijacking control from the programmer-specified CFG. In addition, ProxyCFI also provides protection against non-gadget code reuse attacks (*e.g.*, COOP, where the attack does not leave the CFG of the program but instead enlists the code in a CFG mimicry attack [13]).

3.2.2 Pointer Proxies

To stop control flow attacks, we replace all program code pointers with pointer proxies. A pointer proxy is a random identifier (64-bits in our evaluated implementation), in which pointer proxy P represents an edge from code exit point Y to entry point X. Wherever a code pointer resides in the program (*e.g.*, in a jump table or on the stack as a return address), it is replaced by its corresponding pointer proxy value P. Figure 3.1 illustrates a small code snippet in which the code pointers have been replaced by pointer proxies. As seen in the example, where code pointers would have been stored (*e.g.*, on the stack for a return address), they are replaced with pointer proxies (denoted by a $).

At indirect jumps and returns, the pointer proxy is inspected, and then, using a multi-way direct branch, the appropriate code entry point associated with the pointer proxy is targeted. We call a multi-way direct branch, which matches a pointer proxy and then directly jumps to the associated code target, a *sled*. Direct jumps are not replaced with pointer proxies. Since our threat model assumes that code cannot be written, any direct jump is naturally a write-protected programmer-specified control transfer, and thus, no additional protections are required. Three multi-way branches can be seen in the example in Figure 3.1. The indirect call to *bar()* and *baz()* in function *foo()* is implemented with a multi-way branch that jumps to *bar()* if the proxy *$7743d2ff* is encountered and jumps to *baz* when the pointer proxy is *$1f324a19*. Additionally, both of the returns from functions *bar()* and *baz()* are implemented with a multi-way branch.

Pointer proxies are assigned to code pointers within the context of a single function; thus, the pointer proxies of function X are meaningless to function Y. This powerful feature, which does not impact the usability of pointer proxies, works to thwart a large number of advanced CFG mimicry attacks. These powerful attacks, such as control-low bending [12] undermine CFI by using a code pointer copied from one function context to jump to addresses in some other function without violating CFI constraints. To stop the potential forgery of pointer proxies, all pointer proxy values are defined per-function, and they are assigned at program load time by the pointer proxy verifier as detailed in Section 3.2.4. This aspect of pointer proxy context is shown in Figure 3.1 in the returns of functions *bar()* and *baz()*. While both functions return to the same address (*i.e.*, label *done*), they each use a distinctly different pointer proxy. As such, if each of the functions were to steal each other's pointer proxy and return to it, it would not match any target in the return's multi-way branch, and the program would abort.

3.2.3 Building Code with Pointer Proxies

Building code to work with pointer proxies requires replacing every place in the program that uses a code pointer with a pointer proxy. Code pointers in typical programs are used to *i)* capture a reference to a code entry point, typically when a *switch* statement is implemented with a jump table of code pointers; *ii)* capture a reference to a function entry point, typically when a call through a function pointer or virtual function method invocation occurs; and *iii)* capture a reference to a return point in the program, which is stored on the stack during a function call. Note that in each of these cases, an indirect branch of some form is used (*e.g.*, jump-through-register or return); thus, the compilation of all indirect jumps is the focal point of all pointer proxy activities.

Replacing indirection with pointer proxies. All indirect branches (*e.g.*, switch jump tables, indirect calls, and returns) are replaced with multi-way direct branch sleds. Of course, to know what targets must be tested for in a sled, we must fully anticipate all of the targets of each indirect branch. For locally sourced indirect jumps, such as a switch statement jump table, we can easily anticipate in the current compilation module all of the indirect jump targets. For indirect calls and returns, more work is necessary because the locations (and pointer proxy values) may come from another compilation unit. Consequently, the compilation framework must support whole-program CFG construction (including the call graph). In our prototype implementation in the GNU GCC toolchain, we utilize a two-pass compilation strategy, to first build the whole program CFG and then to compile programs with fully enumerated multi-way direct branch sleds. Details of this compilation strategy are covered in Section 3.3.

At indirect calls, the type of the target function is noted, and when the whole program CFG is constructed, an indirect function call is assumed to possibly happen to only same-typed function and has had its address taken (&). Similarly, the return address sleds target the instruction after all actual and potential calls to the returning function, in which a potential call directly targets the function or indirectly targets the function through a compatible function pointer. This approach works quite well until a program declares a *void ** indirect function call, which could potentially call any function in the program. Fortunately, our optimizations, detailed in Section 3.3.2, perform well to reduce the overall impact of these generic indirect function calls.

When good code pointers go bad. Our approach to control flow integrity replaces code pointers with less powerful pointer proxies. Despite this decrease in capability for code pointers, the programming language (C and C++ for our prototype implementation) may still allow a more full range of capabilities for code pointers. In C for example, code pointers are represented in the language as full-fledged pointers; thus, it is possible to encounter operations on code pointers that deviate from the low-capability pointer proxies, such as pointer arithmetic and casting to and from code pointers. For example, we could manufacture a code pointer to a private function in x86-64 GCC by simply adding the size of the preceding function (in bytes) to its code pointer.

In our prototype GNU GCC C/C++ implementation of pointer proxies, we issue a security warning for these operations and then compile the requested operations into the program. If the program is performing potentially dangerous code pointer operations, this will result in code performing the same operations on pointer proxies, resulting in invalid pointer proxies that cause the program to declare an attempt to break the programmer-specified CFG

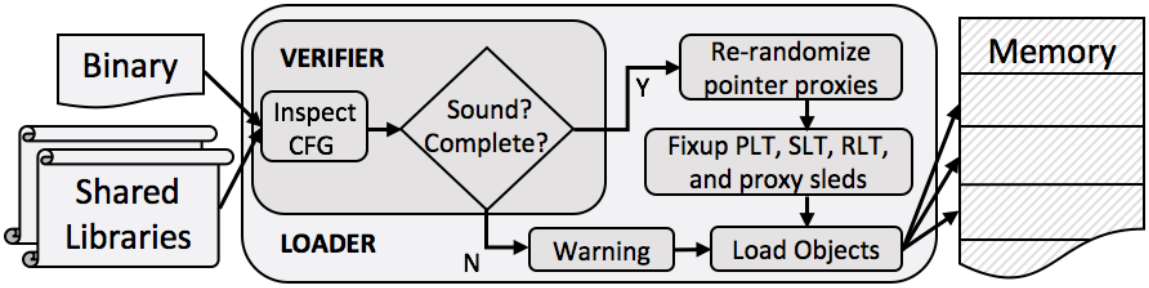


Figure 3.2 ProxyCFI Program Loader. This figure illustrates the process of loading a ProxyCFI compliant program for execution. The program is first inspected by the verifier to ensure its use of pointer proxies covers the entire CFG. The loader then re-assigns randomly selected load-time pointer proxy values to all pointer proxies in the program. This re-assignment coupled with non-readable code pages prevents attackers from building associations between pointer proxies and program address even in the presence of an active read attack.

(when a multi-way direct branch sled aborts). While we were able to create these problems in test programs, none of our benchmark programs suffered from dangerous code pointer manipulations.

setjump() and *longjmp()* require special handling in the compiler because these functions implement a unique user-directed program control flow transition. Together, these functions implement a superset of function pointer behavior, such that a call to *setjmp()* can be the target of any other *longjmp()* in the program. This extra functionality of *longjmp()* can be easily addressed with pointer proxies. Both function pointers and *longjmp()* share an indirect jump, but a *longjmp* will generate a multi-way direct branch, including all of the pointer proxies assigned to the instruction immediately after each call to *setjmp()*. Thus, any tampering in the *setjmp()* control context cannot pull execution off the CFG.

3.2.4 Load-time Program Verifier

It is a key requirement of any ProxyCFI implementation that it covers the entire control flow graph. This ensures that the attacker cannot simply leave the control flow graph and execute instructions with dangerous code pointers and unconstrained indirection. Hence, no benign control flow transfer results in a violation and no deviation from programmer-specified control flow is missed. ProxyCFI maintains this property by verifying that programs utilize only pointer proxies for indirect branches via multi-way direct branches that are fully enumerated by the programmer-specified CFG. If an unexpected pointer proxy is encountered, the program is terminated.

Figure 3.2 shows how a binary or shared library is loaded and validated to only use pointer proxies for control transitions. If a code object passes verification, the verifier

generates load-time assigned pointer proxies, such that an attacker cannot anticipate any pointer proxy values even with an active read attack on the program.

Algorithm 3.1 ProxyCFI Load-time Program Verifier. When ProxyCFI programs are loaded for execution, the following verification algorithm is run to ensure that the program maintains CFG integrity that covers all control flow. The algorithm performs a reachability analyses of the CFG to identify any illegal jumps or uses of indirection.

```
1:  procedure VERIFY(obj)
2:      for all f in obj do
3:          ep ←f.entry_point
4:          while ep ≠ ∅ do
5:              e ← ep.pop()
6:              if e.checked == True then
7:                  continue
8:              else
9:                  br ←scan_for_next_branch(e)
10:                 switch br do
11:                     case Indirect(br) or Invalid(br.target)
12:                         return Fail
13:                     case Direct_Branch
14:                         ep.push({br.target, e.next})
15:                     case sled
16:                         inspect ({sled.proxies})
17:                         ep.push({sled.targets})
18:     return Success
```

The psuedocode for the ProxyCFI code verifier is shown in Algorithm 3.1. The verifier performs reachability analyses on the code object's CFG to validate that it is *i)* free of indirect control transfers and *ii)* all control transfers point to a valid instruction within the current code object or the entry point of another code object for calls. To this end, it performs a breadth-first traversal of the CFG of the code object, inspecting all control transfer instructions. If indirect call/jump or return instructions are encountered in the code object, it immediately fails verification. For direct control transfer (*i.e.*, direct call, direct jump, loop instructions), it analyzes the target address for any possible violations.

For direct jump instructions, the verifier checks that the target address points to a valid instruction within the current code object. For direct function calls, the verifier validates that the target is a valid code object entry point. For multi-way branch sleds replacing an indirect call/jump, the verifier validates that the targets are valid pointer proxies for function entry points. Finally, for multi-way branch sleds replacing function returns, the verifier ensures that all targets follow potential calls to the current function, either directly or indirectly. Once the verifier completes reachability analyses of the control flow graph without failure,

the code is assured to use pointer proxies for all indirect control flow, and thus, it is safe to load and execute. Interestingly, the use of fully enumerated multi-way branch sleds for implementing indirection is precisely the reason why the verifier is able to perform complete reachability analyses of the code. The same would not be possible with unconstrained indirection, since this would require strong assurances as to what code pointers could or could not be created by the program.

3.2.5 Deterring CFG Mimicry Attacks

A mimicry attack [58] on the CFG is one that implements attacker-directed control *without* leaving the programmer-specified CFG. With the introduction of powerful control flow integrity mechanisms, such as CFI [10] and ASLR [5], these non-gadget code reuse-based attacks have quickly grown in number, including counterfeit OOP [13], control flow bending [12] and active-set backward-edge attacks [56]. ProxyCFI can provide protection against these attacks, in particular, through per-function pointer proxy namespaces and load-time pointer proxy assignment.

Per-function pointer proxy namespaces. Traditional full-fledged code pointers represent a code location that is sharable with any other part of the program. It is this property that allows an adversary to copy a code pointer from one function and replay it in an attack on another function, an approach that Counterfeit OOP [13] utilizes to implement method-level code reuse that does not leave the CFG. Pointer proxies deter these copy-based CFG mimicry attacks by defining unique pointer proxy namespaces for each function. Thus, if a function copies the pointer proxy from another function, for example, by searching for pointer proxies up the stack, any attempt to use that pointer proxy will always result in an abort when the multi-way branch sled executes. The only copying of a pointer proxy that would not be detected involves it being in the context of a recursive function. In Figure 3.1, although *bar*() and *baz*() both return to the same location, they each have their own proxy namespace, which have different proxy-to-edge mappings. The differing pointer proxies $ae23afcc and $bc41c823 in the multi-way branches branches implement the per-function namespaces.

Function pointer proxies do not enjoy the same weakened state that return pointer proxies leverage. Function entry pointer proxies may be passed around arbitrarily through parameter lists, return expressions, and global variables. However, our whole-program CFG analyses infrastructure works to only connect indirect function calls to same-typed function entry points that are publicly accessible. As such, wholesale use of mismatching methods, as is

done by Counterfeit OOP [13] and jump-oriented programming [24], are not allowed. These attacks are potentially still possible, but the attacker's agency to create method and function gadgets are limited to similarly typed functions.

For function pointers that are passed as arguments, the function consuming it cannot reasonably know from where the pointer originated. Therefore, the local agreements that enabled pointer proxy variance among functions are not scalably feasible for function pointer proxies. Like code pointers, function pointer proxies rely on a global agreement—a one-to-one mapping between function pointer proxies and the addresses to which they lead.

Return pointers are a temporal agreement between the calling function and the called function concerning the location execution resumes. Traditional return pointers must contain the return address, but pointer proxies may contain any 32 bit value as long as the caller and the callee agree on the meaning of the pointer proxy. There is but one restriction: the pointer proxy must not collide with any other agreements the caller is engaged in. Each function has its own pointer proxy namespace. Formally:

$$ptr_1, ptr_2 \in return\_sled(foo) \Rightarrow ptr_1 = ptr_2 \tag{3.1}$$

Given a pointer proxy used for returning from a function, a function must have only one way to interpret it. In contrast, traditional return pointers have an excessively powerful restriction: given a code pointer, *all functions* must have only one way to interpret it.

Suppose an adversary intends to attack a return edge e on the CFG. Given a non-readable code section, the attacker must first see edge e used before they may attempt a confused deputy attack. Consider the function pointer call in Figure 3.1. Functions foo and *bar* both return to the same location, but they need not share the same pointer proxy for returning. Instead, *main* generates a pointer proxy separately for foo and *bar*. To attack the return in foo, an attacker must first dynamically observe the proxy used: it is insufficient to steal the pointer proxy used in *bar*.

Load-time assignment of pointer proxies. Pointer proxies are re-randomized at load time to further deter mimicry attacks on the CFG (and all other control flow attacks in general). Load-time re-assignment of random pointer proxies prevents offline analyses of code to generate a translation table from pointer proxies to source and target code addresses. Pointer proxy forgery requires prior knowledge of the pointer proxies; thus, enforcing a non-readable code section and load-time assignment of proxies significantly complicates mimicry attacks on the CFG. This is particularly powerful because a pointer proxy is only meaningful in the context of the function that uses it. Hence, to exploit a pointer proxy, the attacker must either share proxies among separate code reuse attacks or write their attack

solely with recursive invocations on the enclosing function. In fact, because pointer proxies are assigned randomly, they cannot be effectively guessed; thus, any mimicry attack on a pointer proxy protected CFG would, in the very least, require an active read attack to expose possible pointer proxy values that have been stored in the data segment.

3.2.6 Shared Libraries with Pointer Proxies

Shared libraries are an attractive target for control flow attacks because they are used among multiple applications. Attacks that target libc, for example, can be reused on any application that links to it. The classic attack is *return-into-libc* [8], wherein the adversary overwrites the return address so that the program returns to an exploitable *libc* function such as *system()*. Shared libraries, in general, are a popular target for control flow attacks as they are loaded for most programs and contain wrappers for system calls. In addition, most shared libraries contain large enough codebases, leaving an attacker with wide selection of gadgets for all classes of code reuse attacks, such as ROP [9], JOP [24], LOP [25] and their variants [59][60].

Shared libraries are built on indirection. Connections into and out of shared libraries must be managed by unshared data or code that is generated dynamically. Returns and indirect calls are natural solutions to entering and exiting shared libraries because they draw on unshared data in the stack and the global offset table, respectively. As such, shared libraries therefore clash with ProxyCFI which works to remove indirection. One solution to securing shared libraries is to forbid them; Intel chose this with SGX [61]. However, we want to retain their advantages: reduced page swapping, simplified version management and facilitated modularity.

Insecure shared libraries utilize unshared data to manage connections between code objects via returns and indirect calls. ProxyCFI compliant shared libraries, however, must use *unshared code* to manage control flow in and out because indirection must be replaced with multi-way branches. While calling a shared library function still traverses the *procedure linkage table (PLT)*, the indirect call within the PLT is dynamically replaced with a pointer proxy sled. At load time, extra space in the caller's address space is mmap'ed for the code that channels control flow on return. Shared library functions have their returns statically replaced with a relative jump down to the unshared multi-way branches.

Our approach to deploying shared libraries with ProxyCFI is illustrated in Figure 3.3. We split the process of returning from a shared library into two stages, which are associated with the *selection linkage table (SLT)* and the *return linkage table (RLT)*, respectively. Two pointer proxies are used to return from a shared library, one for the SLT and one for the

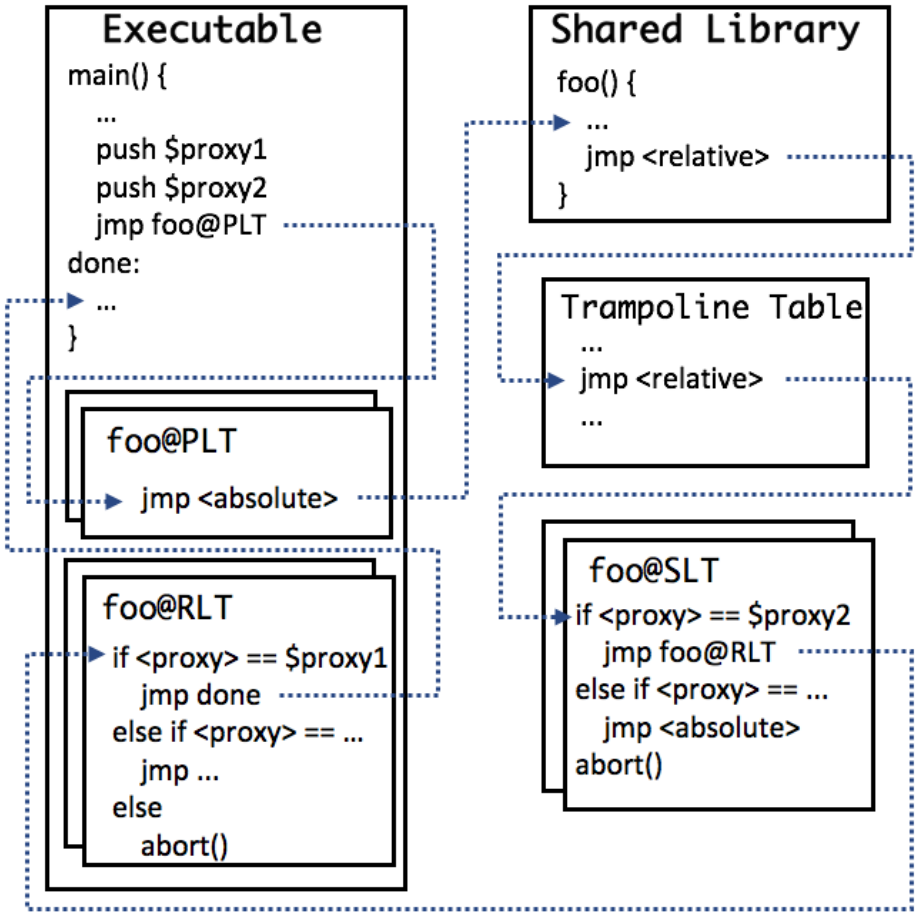


Figure 3.3 Shared Library Control Flow. Shared libraries, which rely heavily on indirect jumps and returns, can be built with proxy pointers. We allocate linkage tables using pointer proxies in the caller's address space, which permit entry and exit from the ProxyCFI compliant shared library.

RLT. For each PLT entry to a shared library function, there is an RLT entry that contains a multi-way branch leading back to each call site in the code object. If shared library function $foo()$ is called from multiple code objects, then each code object will have a separate RLT entry for $foo()$ in its own address space. While the RLT specifies how to return within a code object, the SLT specifies which code object to return to. To accomplish this, SLT entries contain a multi-way branch of absolute jumps directed at RLT entries. Since the size of SLT entries varies based on the code objects that use the shared library, relative jumps down from shared library functions cannot target SLT entries directly. A trampoline table facilitates static generation of the relative jumps by forwarding control onto the appropriate SLT entry.

Using load-time pointer proxy assignment, it is possible to assign proxies for the tendrils into a shared library in a way that creates pointer proxies when the library is first loaded. Moreover, our approach allows the pointer proxies used to enter and exit the shared library

function to be unique to each address space that utilizes the shared library. This feature ensures that an attacker cannot gather pointer proxy information from their own address space and use it to attack a program using the same shared library.

Re-randomization must be performed carefully to allow code pages to be shared. Since SLT multi-way branches are generated at load time, reassigning their proxies will not un-share any code pages. A call from one shared library to another must be moved to an unshared code page since the call is transformed to push two proxies onto the stack (one for the SLT and one for RLT). Re-randomization of proxies requires that the RLT be unshared since all non-empty multi-way branches contain pointer proxies.

3.3 ProxyCFI in GNU GCC

In this section, we detail the implementation of ProxyCFI in the GNU GCC C/C++ toolchain. We present the overall compilation flow, and then dive into the details of the optimizations implemented.

3.3.1 Compilation Flow

ProxyCFI instrumentation involves a two-pass transformation on assembly generated mid-compilation by the existing GCC infrastructure. All sites of indirection are replaced with fully enumerated multi-way direct branches that validate CFG transitions with pointer proxies. Figure 3.4 describes the overall flow of ProxyCFI compilation.

- **Pass 1. CFG Discovery**: Assembly files are parsed for function labels, (direct or indirect) call sites and return sites. Return edges are constructed by observing the target set for each direct and indirect call. Indirect call target sets include only functions that have had their addresses taken and have a matching type signature. Type information on function pointer calls are passed from the GCC frontend to the ProxyCFI compiler core.

- **Pass 2. Branch Enumeration**: Since the CFG contains all transitions between functions, multi-way branch targets are fully enumerated before the second pass begins. For return sites, pointer proxies are selected in the context of the called function and shared with the calling function's indirect call sled. Pointer proxies are generated in this way to deter CFG mimicry attacks (see Section 3.2.5 for details).

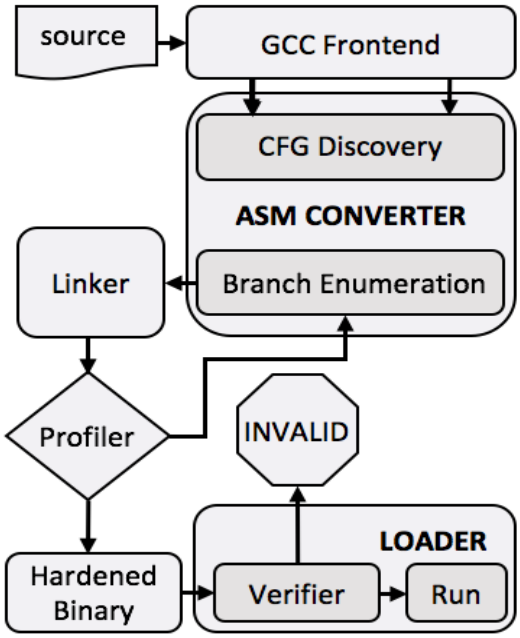


Figure 3.4 ProxyCFI Compilation Flow. The compilation occurs in two passes. In the first pass, the entire CFG of the program is discovered using identifier type and visibility information gathered in a full pass over the program code. In the second pass, all legal program entrypoints are assigned randomly selected pointer proxy identifiers, and all indirect jumps, indirect calls and returns are replaced with fully enumerated multi-way direct branches that translate pointer proxies to direct jumps. The linker resolves all jumps using compiler-generated global identifiers for all entry points. The profiler instruments the code to count the most frequent targets of multi-way branches, which is used for optimization. Finally, all code is passed through the pointer proxy verifier, assigned load-time random pointer proxies and loaded into execute-only pages before execution begins.

After generating a binary, runtime analytics, which are generated by the profiler, are passed back to the branch enumeration phase, at which point the multi-way branch sleds are rewritten with optimizations. Load time invocation of the verifier rewrites all pointer proxies before executing the program, ensuring that attackers cannot use offline analyses to observe pointer proxies before launching an attack.

3.3.2 ProxyCFI Optimizations

Indirect jump and return sleds can become very long, especially for frequently called functions. To address these potential concerns, we implemented two optimizations: profile-guided sled sorting and function cloning.

Profile-guided sled sorting. The main source of performance degradation with ProxyCFI is the overhead incurred by the repeated comparisons used to implement multi-way direct

branch sleds. The number of checks required is directly proportional to the number of legitimate targets for the corresponding indirect control transfer instruction. By analyzing our early implementation, we observed that some multi-way direct branch sleds suffered significant performance degradation. Yet, we also observed that these sleds were highly biased to only a few of the branch targets. Our sled sorting optimization takes advantage of the biased distribution of multi-way branch targets by sorting the order of entries in a sled in descending order of profiled execution count. As shown in Section 3.4, this optimization significantly reduces the average depth a program must traverse into a sled before finding the pointer proxy target.

Function Cloning. While profile-guided sorting of the sleds significantly reduces performance degradation associated with multi-way branch sleds, the improvements are limited for functions with more uniformly distributed sled profiles. To combat this, we adapted function cloning [62] – an optimization that creates specialized copies of functions – as a means to reduce overall sled lengths. For sleds with more uniform distributions, this optimization significantly reduces the performance overhead incurred by executing sleds. Figure 3.5 illustrates function cloning. A function with near uniform sled distribution is cloned (*e.g.*, function *f2* becomes identical functions *f2* and *f2_c1*). Then, half of the call sites to the cloned function are redirected to the cloned function. This has the desirable effect of cutting both the cloned function's and the clone's return sleds in half.

Once a clone function is created, we evaluated the reassignment of multi-way branch sled entries among clones using the following approaches: *i)* evenly distributing the sled entries between clones in descending order of their dynamic execution count; *ii)* assigning the most frequent entries to one clone and least frequent entries to the other to address worst case execution count on least frequently taken branches. The results show that scenario *i)* performs better, so we chose this method as our assignment strategy.

Function cloning also improves the security guarantee provided by ProxyCFI, as it cuts the number of return target addresses at the expense of an increase in code size. For control flow attacks that do not leave the CFG [12][54][13], this optimization would significantly reduce the attacker's agency in selecting CFG edges to exploit. Moreover, gadget dispatcher functions (functions that can overwrite their own return address and have large target sets such as *memcpy()*) qualify for function cloning, which strictly limits the number of legitimate edges for every clone of the function. This observation is contrary to optimizations used for coarse-grained CFI that reduce the incurred performance overhead by merging labels used for checks [63] – which weakens the security guarantee by over-approximating the allowed targets for indirect control transfers.

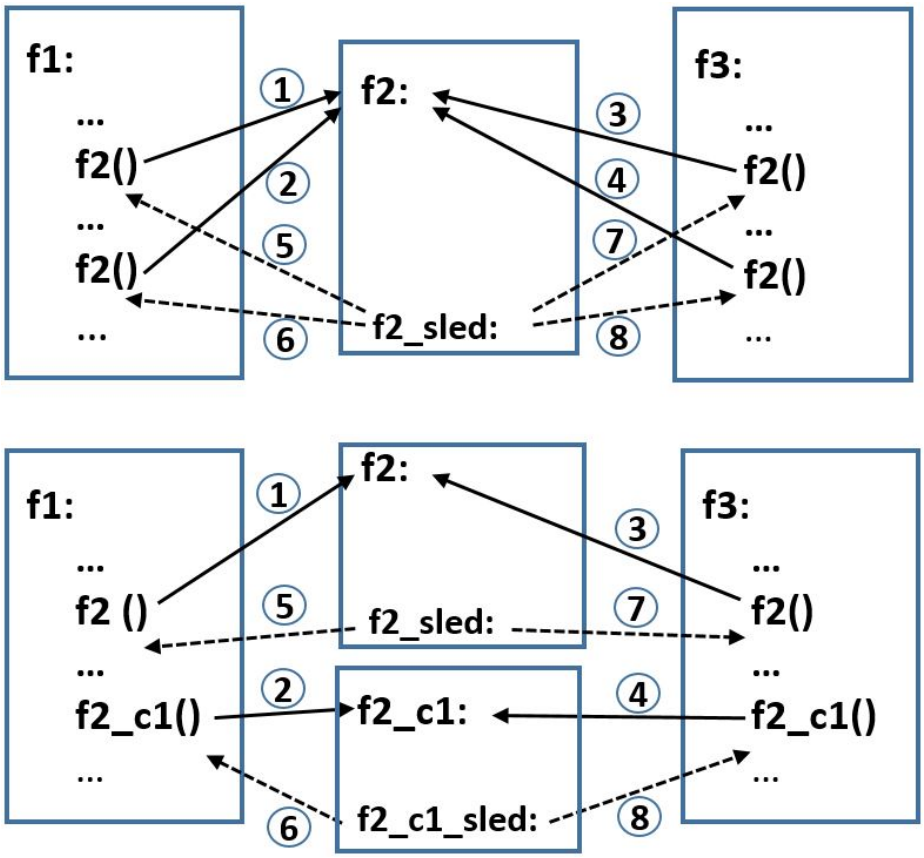


Figure 3.5 ProxyCFI Function Cloning Optimization. Function cloning cuts the number of legitimate edges by a factor of the number of clones. Legitimate edges 6 and 8 from f2_sled are no longer legitimate after cloning. This optimization reduces return sled lengths and raises the bar for CFG mimicry attacks, at the cost of increased code size.

3.4 Evaluation

In this section, we examine the performance and security of ProxyCFI. First, the performance impact of ProxyCFI is assessed by examining the slowdown incurred for many CPU-centric and network-facing benchmarks, with and without ProxyCFI optimizations. We then examine the overheads occurred when using ProxyCFI with shared libraries. To gauge the security benefits, we performed penetration testing with the RIPE control flow attack suite [57]. In addition, we demonstrate the ability to stop a variety of additional real-world control flow attacks, including CFG mimicry attacks.

3.4.1 Evaluation Framework

ProxyCFI build framework. Our ProxyCFI compiler framework was built on GCC version 6.1.0. In our evaluations, we used Ubuntu 16.04 on x86-64. Using x86-64 is essential

in our implementation because our shared libraries rely heavily on relative jumps to preserve code page sharing, which is significantly more efficient in 64-bit x86. We customized *GLIBC's* loader to handle ProxyCFI compliant shared libraries and mark code pages execute-only. Many modern processors have hardware support for execute-only memory. For example, recent Intel CPUs support unreadable code pages using the Memory Protection Keys (MPK) feature.[1] In our prototype implementation, we used this feature to make the code section execute-only (*i.e.*, disabled read/write access).

We built our benchmarks with the *musl* C library. This decision allowed us to sidestep the added complexity of employing pointer proxies on GLIBC, which makes frequent use of GCC-specific virtual `ifuncs` function pointers, which, in turn require special (but arduous) modifications to the GCC compiler backend. In contrast, *musl*'s goal of high portability made it a straightforward port into our compilation framework, and it did not compromise our system's ability to build large projects, such as `redis`.

Benchmarks analyzed. We evaluated the performance and space overhead incurred by ProxyCFI using the SPEC CPU 2006 benchmarks. In addition, we evaluated the overhead on the network-facing application ***redis-server***, running it with the standard *redis-benchmark* with 50 parallel clients and a 3-byte payload. To isolate the performance overhead incurred by ProxyCFI-hardened shared objects, we also ran microbenchmarks for varying shared library sled depths. To evaluate the security guarantees provided by ProxyCFI, we analyzed applications from all the major categories commonly targeted by control flow hijacking attacks, including multimedia processing, Javascript engines, document rendering, network infrastructure and VM interpreters. Specifically, we analyzed the following common attack targets:

MuPDF is a light weight PDF XPS and EPUB parsing and rendering engine. MuPDF versions V1.3 and prior have a stack-based buffer overflow vulnerability (CVE-2014-2013) [64] that results in remote code execution via a maliciously crafted XPS document.

bladeenc is a cross-platform MP3 encoder, which is also used as a daemon for encoding in distributed MP3 encoders/CDDB servers like *abcde*. *bladeenc* has several vulnerabilities that lead to control flow attacks including, CFG mimicry attacks that could be exploited remotely (CVE-2017-14648) [65].

dnsmasq is a DNS forwarder designed to provide DNS services to a small-scale networks, and it is included in most Linux distributions. Versions of *dnsmasq* prior to 2.78 have a stack-overflow vulnerability that enables a remote attacker to send a maliciously crafted DHCPv6 request to hijack control flow on the target system (CVE-2017-14493) [66].

[1] Execute-only memory is also supported on ARMv8 and above.

Gravity is a dynamically typed concurrent scripting language written in C. The Gravity runtime contains a stack-based buffer overflow that leads to remote code execution (CVE-2017-1000437) [67]. We built upon the proof-of-concept exploits provided for these vulnerable applications to test the effectiveness of ProxyCFI in stopping real-world control flow exploits, including CFG mimicry attacks.

3.4.2 Performance Analyses

We ran the SPEC CPU 2006 benchmarks performance analyses experiments on an Intel Xeon Gold 6126 Processor with 24 cores and 32GB RAM running Ubuntu 16.04 LTS Xenial Xerus with Linux kernel 4.15.0-33-generic. For the I/O-intensive network-facing applications, we ran the experiments on an Intel Core i7 5500U, running at 2.40 GHz with 8GB RAM.

Figure 3.6 shows the performance overhead incurred by ProxyCFI instrumentation, with the results summarized in Table 3.3. For compute-intensive applications, the näive implementation's performance overheads are non-trivial, since these programs have high average sled depth. Average sled depth is a measure of how many pointer proxy tests are required in a sled, on average, before a direct branch is taken. Ideally, we would like this value to be close to 1 to lower the performance overhead for ProxyCFI. For applications with heavy use of function calls, such as *perlbench*, *gobmk* and *sjeng*, the performance degradation for the unoptimized implementation is more pronounced, having a average return sled depth of 27 for *perlbench*.

With optimizations, the average sled depth drops dramatically, as do the performance overheads. For example, *perlbench* benefits significantly from profile-guided sled sorting optimization. *h264ref* also benefits significantly from optimizations, as it makes heavy use of generic function pointers with indirect functional call sleds having up to 855 entries, of which only two are frequently targeted. *gcc*, on the other hand, makes considerable use of both function calls (average sled depth of 32) and generic function pointer (with an average sled depth of 26 for indirect calls). The performance benefit of the function cloning optimization is more visible on *gcc*, as the probability distribution of taken branches falls off slower than the other applications. For network-facing applications, the performance overhead is insignificant due to their I/O-bound nature. The average performance overhead of ProxyCFI on *redis-server* is 0.25% and overall incurs an average overhead of 0.93% for all of network-facing applications we evaluated. Table 3.2 shows the break down of the results of running *redis-server* with the standard *redis-benchmark* using 50 parallel clients and a 3-byte payload.

Table 3.2 Results of running *redis-benchmark* on ProxyCFI compliant *redis-server* versus unhardened baseline

Command	Baseline (request/sec)	ProxyCFI (request/sec)
PING_INLINE	12320.9	12115.34
PING_BULK	12881.67	12926.58
SET	12469.83	12158.05
GET	12941.73	13010.67
INCR	10514.14	11189.44
LPUSH	12227.93	12997.47
RPUSH	11592.86	11828.72
LPOP	12659.83	12255.46
RPOP	12804.1	12604.8
SADD	12218.96	12055.46
HSET	12023.57	11872.7
SPOP	11855.36	11552.39
LPUSH (needed to benchmark LRANGE)	12968.49	12600.12
LRANGE_100 (first 100 elements)	6506.82	6325.19
LRANGE_300 (first 300 elements)	2788.99	2690.05
LRANGE_500 (first 450 elements)	2403.4	2211.26
LRANGE_600 (first 600 elements)	1730.2	1652.59
MSET (10 keys)	10409.08	9959.79

Figure 3.7 shows the percentage increase in the binary size as a result of pointer proxy instrumentation, both with and without optimizations. On average the code size grows by 49% for our benchmarks with the worst case of 121% for *h264ref*, due to the large amount of instrumentation required for its generic function pointers. Finally, Figure 3.8 shows the impact of ProxyCFI verification and load-time proxy randomization on program load times. As shown in the graph, the load time impacts are minimal, adding at most 1 second to the load time of the largest benchmark.

Shared library performance. Shared libraries have a strong reliance on indirect calls and jumps, which poses a challenge to a ProxyCFI system because all control flow indirection must be replaced with multi-way direct branches. In our analyses, we measure the cost of our shared library support infrastructure by microbenchmarking entries and exits to shared libraries, ultimately comparing the cost to unprotected shared library calls. The average percent slowdown for a shared library call using optimized ProxyCFI compilation is 1.48% and 2.31%, respectively, for the best and worst-case average sled hit depths observed in our benchmark experiments.

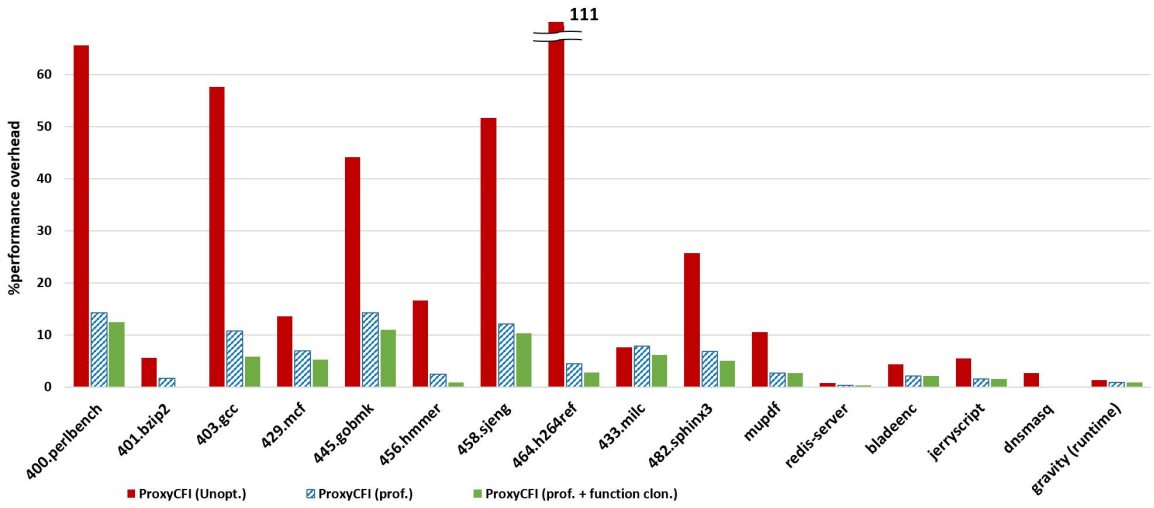


Figure 3.6 Performance Overhead of ProxyCFI. ProxyCFI (Unopt.) shows the performance overhead for unoptimized, while ProxyCFI (prof.) and ProxyCFI (Prof. + function clon.) show performance with varied optimizations applied. The numbered programs on the left are computationally intensive programs from SPEC2006, while the unnumbered programs on the right are I/O-intensive benchmarks.

		Unprofiled	Profiled	Profiled + cloning
Performance	SPEC	39.95%	8.14%	5.95%
	Network	4.15%	1.21%	0.93%
	Overall	25.9%	5.43%	4.09%
Binary Size	SPEC	51.9%	51.9%	61.5%
	Network	44.4%	44.4%	50.6%
	Overall	49.1%	49.1%	57.4%
Sled Hit Depth	Fptrs	18.54	1.33	1.31
	Returns	7.03	1.26	1.24

Table 3.3 Summary of ProxyCFI Overheads. This table summarizes the performance and code size impacts of ProxyCFI. The top line lists the slowdown for ProxyCFI instrumented programs running on an Intel Xeon CPU. The second line lists the impact on code size due to ProxyCFI, with and without optimizations. The bottom line of the table shows the average depth into multi-way branch sleds at jumps, with and without optimizations.

Load-time overhead. We measured the load-time overhead incurred by pointer proxy randomization at load time. We observed that it has approximately linear relationship with code size, consistent with previous works that perform load-time randomization [39] [68]. Figure 3.8 shows the load-time overhead incurred for the benchmarks used for our evaluation, the longest being 1200 ms for *redis-server*. The average load time is 0.98 ms per

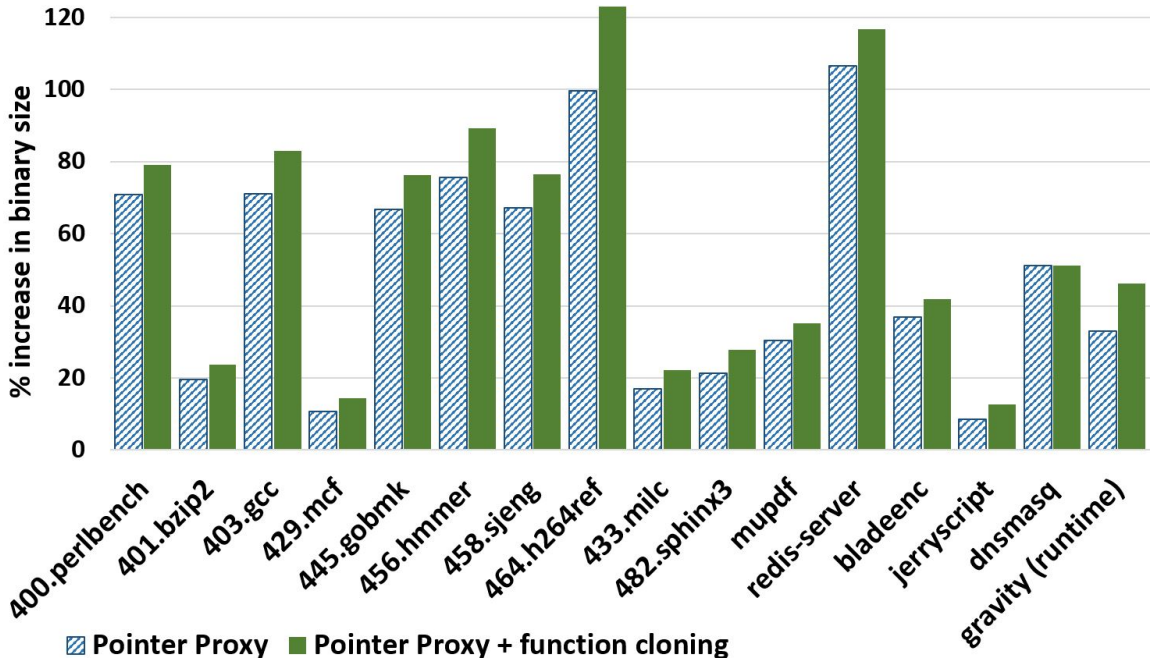


Figure 3.7 ProxyCFI's Increase in Code Size. This graph shows the impact of ProxyCFI on code size. Code size increases are shown with respect to the original program with code pointers and no control flow protections. The blue bars (left) represent unoptimized ProxyCFI programs, while the green bar (right) represents optimized ProxyCFI programs. The function-cloning optimization introduces code into the program binary, thus, its increased performance and security comes with a slight increase in code size.

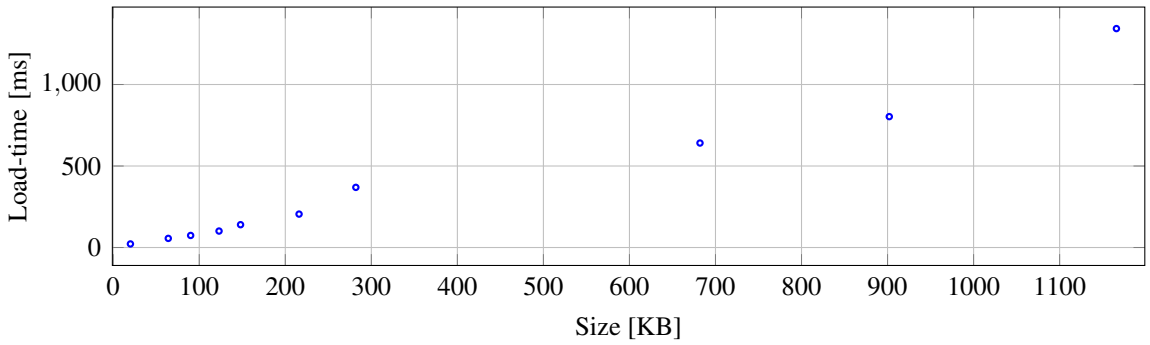


Figure 3.8 ProxyCFI's load-time overhead vs. code size This graph shows the impact of pointer proxy randomization on load-time. It has approximately linear relationship with code size, the longest being 1200ms, consistent with previous works that perform load-time randomization.

KB of code size, which is comparable to Binary Stirring [39] and O-CFI[68].

3.4.3 Security Analyses

To assess the security strength of ProxyCFI, we first examine its ability to stop control flow attacks in the RIPE attack suite, then we examine to what extent ProxyCFI can stop real-world control flow attacks, including CFG mimicry attacks.

Penetration testing with RIPE RIPE is a control flow attack testbed that generates attacks by permuting five dimensions of attack: location (*e.g.*, stack, heap, ...), target (*e.g.*, return address, function pointers, ...), overflow technique (*e.g.*, direct/indirect) and function of abuse (*e.g.*, memcpy, ...) [57]. Native RIPE targets 32-bit x86 code; thus, with the help of a recently implemented low-fat pointer extension [28], we ported the RIPE test suite to x86-64. Our port supports the following five dimensions: location, target (excluding *setjmp()* and *longjmp()*), overflow method and overflow type. Permuting all the aforementioned RIPE dimensions will result a total of up to 850 unique tests. With all built-in defenses (ASLR, DEP and stack canaries) disabled, 264 attacks succeed on our baseline system (Ubuntu 16.04). **With ProxyCFI protections, 100% of the RIPE attacks are stopped.** In addition, ProxyCFI was able to detect the exact point at which attacks escape the CFG.

Real-world vulnerabilities. To evaluate the effectiveness of ProxyCFI against real attacks, we included recent attacks reported on the National Vulnerability Database (NVD) [69] in our evaluation. We examined two specific types of exploits: making an application crash (to cause Denial of Service attacks) or remote arbitrary code execution attacks. Using ProxyCFI we were able to detect the first variety of attacks by enforcing the dynamic execution flow which caused the crash, and we completely stopped the later by detecting tampering to any pointer proxy. We adopted the vulnerabilities reported on NVD into our benchmarks and introduced exploits that perform control flow attacks including CFG mimicry attacks.

With ProxyCFI, we were able to stop all of the following real-world attacks, including CFG mimicry attacks on *Bladeenc*. None of the tested vulnerabilities were able to inject code locally or remotely, and all of the denial of service attacks declared attacks immediately upon invocation. In testing, we found that the declared violations enabled us to quickly identify the root cause of the vulnerability. We analyzed four attacks.

MuPDF has a stack-based buffer overflow vulnerability in the $xps\_parse\_color()$ function that performs an unchecked *strcpy()* of a user supplied (via XPS input) array to a fixed size buffer[64]. The exploit uses this bug to overwrite the return address and jump to an ROP gadget. With ProxyCFI, we were able to detect the stack pivot based on the corrupted pointer proxy.

Bladeenc's command line parser uses unchecked calls to *strcpy()* to copy parameters to a 256-byte buffer that is exploited for arbitrary code execution by using a carefully crafted command line arguments[65]. The exploit corrupts a function pointer to jump to another function that is also in its legal target set to hijack control flow via a CFG mimicry attack. We were able to detect the exploit when trying to jump using a forged pointer proxy (which was interpreted as invalid pointer proxy from the source address).

Dnsmasq has a vulnerability caused by an unchecked use of *memcpy()* in the *dhcp6_maybe_relay()* function to a 16-byte field of the variable *state*. This bug allows an attacker to perform inter-object overflow to perform ROP attack. Using ProxyCFI, we were able to detect all of the exploits.

Gravity contains a stack-based buffer overflow in the function $operator\_string\ \_add()$ which can be used to write past the end of a fixed-sized static buffer to achieve code execution. The exploit uses this vulnerability to overwrite a return address using a malicious Gravity script. For the ProxyCFI hardened version the attack was detected when the exploit tried to make an indirect jump based on forged pointer proxy.

3.5 Related Work

Memory safety. Memory corruption attacks have been often used to hijack control flow, either by injecting code or reusing existing code. Code reuse attacks, including ROP [9], JOP [24] and return-into-libc [8], are particularly powerful as they defeat even standard protections deployed today. Data execution prevention [7] [6] is sidestepped entirely as reuse attacks need not inject code. While in theory address space layout randomization (ASLR) [5] should stop address forgery, only a single leaked pointer is enough to compromise all benefits of randomizing code addresses, which can be accomplished with a buffer over-read [47]. Comprehensive memory safety techniques, such as Softbound [27], can completely eradicate memory exploitation, but they suffer from high overhead or compatibility issues, deeming them as yet impractical for widespread adoption.

PointGuard [42] protects pointers by encrypting them on storage and decrypting on dereference. To achieve low overhead, the defense uses XOR-encryption of pointers with a global key, which makes it possible for an attacker to exploit the cryptography. They can do this by partially overwriting pointers and performing brute-force relative distance attacks to forge pointers that differ only in the least significant bytes. However, a cryptographically secure software-based version is prohibitively expensive, preventing widespread use.

Control flow integrity. A new wave of practical defenses has emerged with a focus on validating that execution adheres to a static, programmer specified CFG. Control flow integrity (CFI) [10] was the first of these CFG defenses. The defense inserts checks before indirect branches to ensure that all indirect control transfers are within the statically discovered CFG. Various coarse-gained variants have relaxed CFI constraints to achieve practical solutions through both software and hardware approaches[11][50][51][52]. CCFIR [11] uses a load-time randomized springboard section to redirect all indirect control flow transfers, which have been bypassed by a successive work [53]. Intel CET [51] provides rudimentary hardware protection for forward edges through its indirect branch tracking by enforcing coarse-grained CFI, which restricts the targets of indirect branch instructions (indirect call/jump) to entry points of basic blocks. Microsoft CFG enforces a weak form of CFI by restricting indirect function calls to function entry points [52]. Unlike these coarse grained CFI techniques, ProxyCFI provides fine grained protection, and also affords protection against CFG mimicry attacks.

CCFI [14] is a fine-grained CFI technology that protects code pointers by storing hash based message authentication code (MAC) alongside code pointers and checking the MAC before indirect branches. While CCFI can protect against CFG mimicry attacks, its high performance overhead (52% for SPEC'06) will undoubtedly limit its applicability in production environments. Like CCFI, ProxyCFI provides fine-grained control flow protection, while incurring significantly lower overheads (only 5.9% average slowdown for SPEC'06).

Other control flow integrity works have proposed to completely remove instructions employed for control flow hijacking attacks. Return-less kernels [63] avoid use of *ret* instruction by replacing them with a lookup into a static return table, which provides protection solely against return-based attacks. Control-data isolation (CDI) [70] rewrites both forward and backward edges with exclusively direct branches. CDI would conceivably constrain execution to the programmer-specified CFG, if it were to verify that all binaries adhered to CDI compilation requirements. But since the approach still uses code pointers to identify program pointers, the approach is readily attackable with control flow attacks that do not leave the CFG, such as Counterfeit OOP [13]. We adopt CDI's approach of excising indirection from the program, but we back it up with a load-time verifier that guarantees the use of pointer proxies for all indirect control flow. Moreover, ProxyCFI addresses CFG mimicry attacks by replacing code pointers with pointer proxies that utilize per-function namespaces, which are assigned at program load-time to execute-only memory.

Readactor [71] replaces code pointers in data with execute-only trampolines to their targets. However, hidden functions imported from other sub-modules can be invoked if trampoline addresses are even partially disclosed [72]. Code-Pointer Integrity (CPI) [33]

provides memory safety for code pointers by storing them (and pointers that point to them) in a safe region, then instrumenting and bounds-checking all accesses to that region. CPI requires allocation of a safe region inaccessible to an attacker which is achieved by segmentation on 32-bit x86 systems and by information hiding on x86-64 systems [73]. ProxyCFI does not require any special data region protections. Moreover, ProxyCFI begins to chip away at the power of CFG mimicry attacks through the use of per-function pointer proxy namespaces and load-time assignment of pointer proxies.

3.6 Chapter Summary

While significant effort has been exerted to shut down control flow attacks, their existence and value persists today, even 40 years after the first buffer overflow attack. With ProxyCFI, we take the novel approach of replacing all of a program's code pointers with the much less powerful pointer proxy. A pointer proxy is a random identifier representing a specific program entry point from the context of a specific function. A control transfer with a pointer proxy utilizes a multi-way direct branch that fully anticipates all of the potential jump targets. As such, ProxyCFI provides much resistance to advanced control flow attacks because it is difficult to forge/swap pointer proxies to mimic a legitimate CFG transition. Our implementation of ProxyCFI is built into the GNU GCC C/C++ compiler toolchain, such that all code pointers are replaced with pointer proxies, including those contained within shared libraries. analyses of our optimized pointer proxy implementation reveals that they introduce minimal slowdown, only an average 4% slowdown with optimizations across a wide range of benchmarks. Moreover, security analyses of ProxyCFI shows that it stops all of the control flow attacks we tested, including 100% of the attacks in the RIPE x86-64 attack suite and a wide range of real-world attacks, including CFG mimicry attacks.

Chapter 4
Runtime Stack Layout Randomization

4.1 Data-Oriented Attacks and Defenses

Despites decades of security research, memory corruption still poses a great threat to software systems. This results from the fact that a large amount of code has been written with memory unsafe languages like C and C++, making them vulnerable to memory corruption attacks. Using memory corruption, attackers deploy control-flow attacks in which the execution flow of a program is manipulated to execute code sequences unanticipated by the programmer, with the ultimate goal of circumventing system security measures.

Due to the prevalence and power of control-flow attacks, various mitigations have been proposed, such as Control-flow Integrity (CFI) [32], which enforces the runtime execution path of a program to adhere to the statically determined Control-Flow Graph (CFG), and Code Pointer Integrity (CPI) [74], which provides memory safety for code pointers. These techniques have been shown to be effective at confining programs to the programmer-specified control flow graph. However, widespread adoption of control-flow attack protections has resulted in attacks that corrupt non-control data to perform malicious operations. Non-control data attacks do not violate the constraints imposed by these defenses as they do not violate control-flow of the program, rather they reuse existing control-flow to manipulate program data. Moreover, recent studies have shown that Turing-complete computation capabilities can be achieved without leaving the statically determined CFG [12] or without modifying code pointers [75]. Control-flow bending bypasses CFI protections by swapping target addresses of an indirect branche with another valid address from the same branch. Data-Oriented Programming (DOP)[75] enables an attacker to execute a sequence of instructions within the legitimate control flow of the program by repeatedly corrupting non-control data. In this thesis, we broadly term any attack that provides a programming capability without leaving the programmer-specified CFG as a *data-oriented programming (DOP)* attack.

Address randomization defenses, *e.g.*, address space layout randomization (ASLR), can be used as a first line of defense against DOP attacks. However, information leaks are increasingly being utilized to bypass these defenses, including fine-grained and runtime based randomization techniques [21]. Information leaks coupled with attackers' knowledge of program internals can successfully bypass state-of-the-art randomization techniques and allow an attacker to launch a successful DOP attack, as we will show in Section 4.2.3.

Most DOP gadgets take advantage of the deterministic nature of the stack layout of programs to grant an attacker the ability to corrupt operands used by gadgets. Stack layout randomization, which has already been proposed, could be a powerful tool to stop DOP attacks. However, those techniques fall short in the presence of memory disclosure by relying on one-time static randomization or coarse-grained random padding. Additionally, it is a requirement to have a true random source at runtime for randomizing the stack, as it can be expected that an attacker will have access to the memory variables used to drive pseudo-random number generation.

In this thesis, we evaluate the effectiveness of previously proposed stack layout randomization techniques at stopping real-world DOP exploits. We show that previous stack-layout protections can be easily overcome by DOP attacks. To address this deficiency, we present Smokestack, a runtime stack-layout randomization technique that randomizes function stack layout at each invocation, using a true random permutation selection that is protected against memory disclosure attacks. Using these defenses, Smokestack is able to thwart proposed and real-world DOP attacks.

Objectives and Contributions. Our objective is to develop a runtime solution resilient to DOP attacks, *i.e.*, attacks that manipulate program execution but do not leave the programmer-specified CFG.

First, we assess the effectiveness of prior stack layout randomization schemes at stopping data-oriented attacks and enumerate their limitations. To this end, we developed a real-world DOP attack based on a recently disclosed vulnerability [76] that is able to achieve a Turing-complete computation capability despite the constraints imposed by previous stack-layout randomization schemes.

Then, we alleviate the limitations of prior stack randomization techniques by presenting a novel runtime stack layout randomization solution, called Smokestack [77], which is capable of stopping stack-based data-oriented attacks. Smokestack randomizes the stack layout of functions for every function invocation, thereby thwarting attacker attempts to discover stack frame layouts. Smokestack implements true-random selection of stack layout permutations that cannot be anticipated, even by attackers with full control over data memory.

Finally, we implemented Smokestack in the LLVM compiler framework. Our implementation provides a secure random permutation, at function invocation, using an intrusion-resistant pseudo-random number generator (cased on the Intel AES-NI instruction set extensions), which is seeded from a true random number source. We present a comprehensive performance evaluation of the SPEC 2006 benchmarks and additional previously vulnerable applications. In addition, we assessed the effectiveness of Smokestack at stopping data-oriented attacks using real-world DOP attacks.

The remainder of the chapter is organized as follows. Section 4.2 presents background on data-oriented attacks and assesses the strength of previous stack layout randomization techniques. Section 4.3 presents details of our proposed runtime stack layout randomization scheme. Sections 4.4 details our LLVM-based implementation. Section 4.5 presents a performance and security evaluation of our prototype system. Finally, Section 4.7 concludes the chapter.

4.2 Background

Attackers can exploit memory corruption vulnerabilities in type-unsafe languages like C and C++ to control vulnerable programs. These vulnerabilities are commonly exploited in a control-flow hijacking attack, in which an attacker uses memory errors to corrupt control data, such as a function pointer, return address, or C++ virtual function table, to eventually hijack the control flow of the program. A wide range of methods for control-flow protection have been proposed. These include enforcement based techniques like CFI [32], CPI [74] as well as randomization techniques like timely randomization of code pointers (TASR) [78]. These mitigations either prevent corruption of control data [74] or stop indirect jumps from leaving the programmer-specified CFG.

With the introduction of powerful control-flow protections like CFI [32], the next avenue for an attacker is using a memory error to overwrite non-control data to cause non-control data attacks [15], which have been shown to cause detrimental effects, such as the leaking of secret keys (HeartBleed) [16]. In non-control-data attacks, the execution of the program adheres to a valid control-flow path in the CFG of the program; however, the data and how it is manipulated is controlled by the attacker. Chen et al. [15] demonstrated that non-control-data attacks can be used to overwrite sensitive data used for decision-making and can cause leakage of sensitive data or cause privilege escalation by overwriting variables used in authorization decisions. In its more generalized form, attackers have shown that Turing-complete computations with rich expressiveness can be achieved by manipulating

only non-critical data [12].

4.2.1 Data-Oriented Programming

Data-oriented programming (DOP) attacks work to corrupt non-control data to execute sequences of instructions within the program using attacker-controlled operands. Each sequence of instructions, called a DOP gadget, performs a single operation that contributes towards the overall attack payload. DOP attacks can achieve Turing completeness by chaining DOP gadgets together through the control of a vulnerable loop, called a DOP gadget dispatcher, that is enclosing multiple DOP gadgets.

A few techniques have been proposed to mitigate non-control data attacks, typically relying on protecting only sensitive data, *e.g.*, kernel data [18] and programmer-annotated critical data [79]. However, proposed mitigations have been mostly ineffective. Hu *et al.* [75] demonstrated that DOP attacks can bypass state-of the art defenses like ASLR to perform malicious operations on real-world applications.

4.2.2 Previous Stack Randomization Efforts

DOP attacks' strong reliance on the manipulating stack variables suggests that previously proposed stack layout randomization efforts may provide a groundwork to stop DOP attacks. Prior works in stack randomization perform one or more of the following transformations:
Stack base address randomization. This transformation randomizes the base address of the stack by allocating random-sized padding at the beginning of the program to make the absolute address of stack objects unpredictable [80, 5, 81].
Random padding at function entry. This transformation adds a random padding at the beginning of functions to randomize the relative alignment between stack frame allocations. Forest et al. [81] proposed adding a random padding before stack frames of functions with buffer variables at compile time. They use the size of the stack frame (greater than 16 bytes) to identify functions containing buffer variables. For every stack frame allocation greater than 16 bytes, their technique adds one of the 8 possible paddings (8, 16, ..., 64 bytes) randomly.
Static stack layout randomization. This transformation permutes stack object allocations in a function at compile time to randomize the relative distances between objects in a stack frame [80].

The main weakness of prior stack layout randomization schemes is that they focus only on protecting against the corruption of code pointers in the stack, which requires knowledge

of the absolute distance between the vulnerable buffer and code pointer of interest. However, DOP attacks only require relative distance to variable of interest, a local variable used in a DOP gadget and a DOP gadget dispatcher for a successful attack. Consequently, to assess the effectiveness of previous stack randomization efforts in stopping DOP attacks, we developed a proof-of-concept DOP exploit for a recently disclosed vulnerability in *librelp* logging library (CVE-2018-1000140) [76]. In the following section, we present the details of the attack.

4.2.3 Bypassing Previous Stack Randomization Efforts

The vulnerability in *librelp* is caused by improper use of *snprintf()*. The C library function *snprintf()* writes a null terminated series of characters and values to a non-zero sized buffered and returns the number of bytes that would have been written assuming there was sufficient space, excluding the terminating null byte. It is a common coding mistake to use *snprintf()* in a loop, assuming it returns the number of bytes actually written. If an attacker manages to control the size of the string to be written on the boundary of the buffer, successive iteration of the loop will grant the attacker a non-linear overflow of the buffer, which can bypass protections, such as stack cookies. Listing 4.1 shows the vulnerable code in *librelp*.

relpTcpChkPeerName() checks valid Subject alternative names (SANs) within a X.509 certificate for a peer name until it finds a match. While doing so, it copies all SANs checked so far to a buffer for error reporting. Our proof-of-concept attack exploits the stack-based buffer overflow in the *relpTcpChkPeerName()* function, as shown in Listing 4.1, to construct a DOP gadget dispatcher and series of DOP gadgets. This is achieved by repeatedly corrupting local variables of functions in the call hierarchy used for controlling a loop and performing operations in the socket initializing function —*relpTcpLstnInit()*. Using static analysis, we discovered gadgets for *MOV, DEREFERENCE* and *STORE* operations. Moreover, we were able to de-randomize statically randomized stack layout, random stack padding and ASLR by taking advantage of the semantics of the underlying program.

```
1 relpTcpChkPeerName(..., gnutls_x509_crt_t cert){
2   ...
3   char szAltName[1024];
4   char allNames[32*1024]; /* for error reporting*/
5   bFoundPositiveMatch = 0;
6   iAllNames = 0;
7
8   iAltName = 0;
9   while(!bFoundPositiveMatch) {
```

```
10      szAltNameLen = sizeof(szAltName);
11      gnuRet = gnutls_x509_crt_get_subject_alt_name(
12              cert, iAltName, szAltName,
13              &szAltNameLen, NULL);
14      if(gnuRet < 0)
15        break;
16      else if(gnuRet == GNUTLS_SAN_DNSNAME) {
17      ...
18        /* stack based buffer-overflow */
19        iAllNames += snprintf(
20                allNames+iAllNames, sizeof(allNames)-
21                iAllNames, "DNSname: %s; ", szAltName);
22        relpTcpChkOnePeerName(pThis, szAltName,
23                        &bFoundPositiveMatch);
24      }
25      ++iAltName;
26    }
27    ...
28
29  done:
30    return r;
31  }
```

Listing 4.1 Vulnerable function in *librelp* logging library.

We exploited this vulnerability by supplying a maliciously crafted X.509 certificate, containing more than 32KB of "subject alt names", to a GnuTLS enabled RELP logging service. By manipulating the size of the string for "subject alt name" on the 32KB boundary, we were able to vary the gap precisely enough to control which part of the stack to overwrite. This is essential for our proof-of-concept attack, as it enables the attack to avoid unintended corruption of adjacent stack resident data, which might lead to a crash.

Bypassing static stack layout randomizations To bypass static stack layout randomization schemes, an attacker can perform a read attack and then infer the layout of the stack by analyzing its contents. In our proof-of-concept exploit, we de-randomized the stack layout of the program by looking for the location of the local variable *hints*, which is a *struct addrinfo* instance that gets printed to *stdout* in case of an address error. Then we use the *snprintf* vulnerability in the vulnerable function to repeatedly overwrite the local variables used for the DOP gadgets and gadget dispatchers to perform our DOP attack. Using this approach, we could easily bypass all previously proposed one-time permutation and padding-based stack-layout randomization schemes.

4.3 Smokestack Runtime Stack Layout Randomization

4.3.1 Design Objectives

Our main objective is to provide a practical mitigation technique to stop stack-based non-control data attacks. To achieve this goal, our solution has to meet the following requirements:

- Provide a runtime stack randomization solution resilient to memory disclosure and pointer leaks.
- Have low performance overhead on both CPU-bound and I/O-bound applications.
- Be compatible with legacy code. This requirement includes source and binary as well as modular support to enable gradual migration of code.

4.3.2 Threat Model

Prior stack layout randomization schemes only consider control-flow attacks and hence rely on the assumption that obfuscating the absolute address of stack resident data is sufficient to stop runtime attacks. Even though they are shown to mitigate control-flow attacks, purely data-oriented attacks are not stopped by these protections. Section 4.2.3 shows that an attacker can use DOP attack to undermine static stack layout randomization and random padding schemes.

In this thesis, we assume a powerful attacker who is able to read/write all writable data memory. However, we assume the attacker is unable to write to non-writable data/code sections and registers used by our instrumentation code. In all, we consider a strong adversary capable of:

- Bypassing protections, such as ASLR, using memory disclosure vulnerabilities in the program to get full read access to all code pages mapped in the address space of the program.
- Exploiting memory vulnerabilities and using the semantics of the underlying program to reverse engineer a randomized stack layout of a function based on a disclosed stack frame data, which allows the adversary to instantiate a runtime attack on future calls of the same function.
- Performing a brute-force attack with a finite number of attempts before being detected by the system.

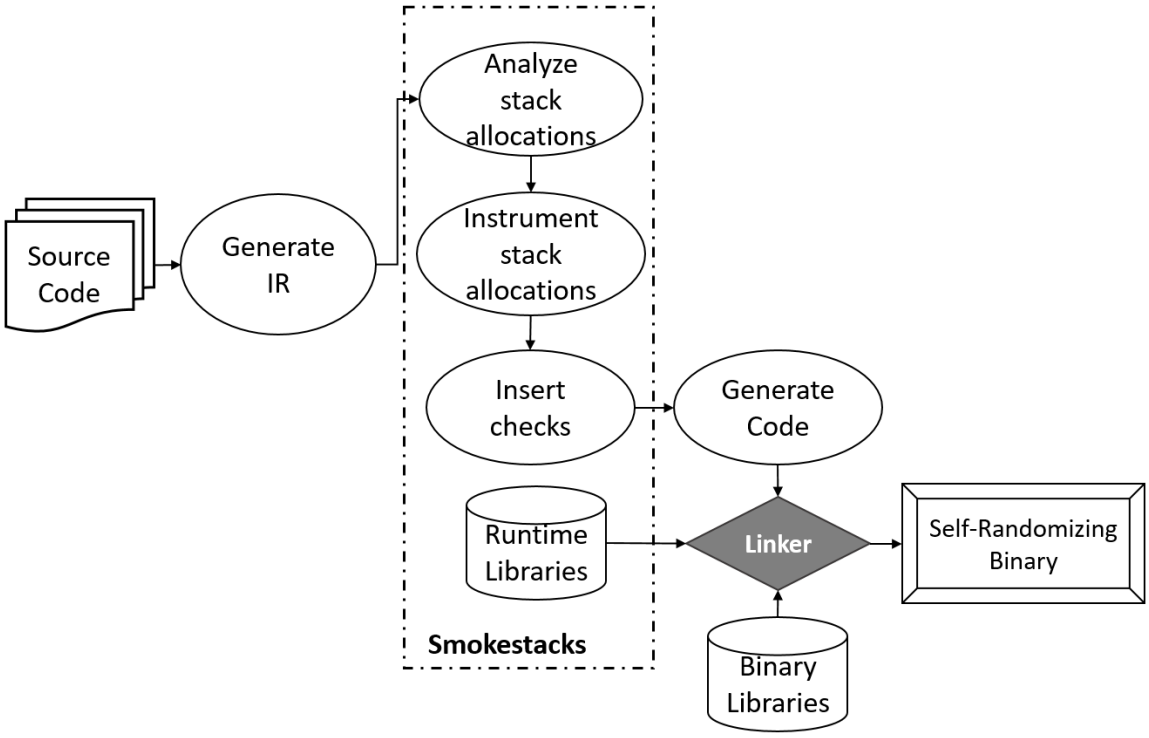


Figure 4.1 Smokestack system Overview. Overview of our runtime stack layout randomization scheme with the components of Smokestack highlighted.

4.3.3 Overview of Smokestack

The primary means to perform non-control-data attacks is to identify local stack variables or registers from the caller that are saved on the stack at the entry of a function and restored before it returns. Then, successive steps of the attack exploit memory corruption vulnerabilities to control the identified local variables to execute the attack payload using the instructions in the vulnerable program. Thus, our protection needs to ensure the absolute address of the variables as well as the relative distance between the changes with every function invocation. Smokestack achieves both goals by dynamically deciding, at the function entry, the ordering, relative distance and alignment between all stack resident objects.

Smokestack performs allocation of stack frames with live re-randomization while retaining all the desirable features of stack allocation, such as automatic deallocation of stack objects for all possible control-flow paths. It achieves this by replacing each stack allocation in a function with a slice into the total allocation, where its location within the total allocation is decided dynamically at the function entry, based on true-random perturbation of the local variables.

Figure 4.1 shows the overview of Smokestack infrastructure. To avoid the performance overhead associated with computing a permutation at runtime, Smokestack embeds a read-

only permutation box, P-BOX, that contains all the possible permutations for all functions in in a shared library that gets dynamically linked with Smokestack-hardened programs. Section 4.3.5 presents optimizations we employed to reduce the associated overheads. The details of the Smokestack compilation and runtime follow.

4.3.4 Discovering Stack Allocations

In this phase, we identify the stack frame allocations for all functions in the program. This includes gathering the type and alignment requirement of for each function's resident objects. We then use this meta data to generate possible permutations of its stack based allocations and the total allocation considering the alignment requirements of all objects. This step requires adding padding to fulfill the alignment requirements of allocations in every possible permutation, which also contributes towards the finally entropy of our randomization.

Algorithm 4.2 Smokestack Permutation Generator. This algorithm generates all the possible permutations of stack allocations within a function.

```
1:  procedure ALIGN(index, alignment)
2:      if index%Alloca.alignment == 0 then
3:          return index
4:      else
5:          return (index / alignment + 1)* alignment
6:  procedure PERMUTE(F)
7:      P_Table ← ∅
8:      N ← Count(F.Alloctions)
9:      for p_index in 0 to N! do
10:         Allocas ← F.Allocations()
11:         temp ← p
12:         curr_index ← p
13:         for a_index in 0 to N! do
14:             Allocas ←F.Allocations
15:             e ←temp/(N − a_index)!
16:             temp ← temp%(N − a_index)!
17:             index ← ALIGN(index, Allocas[e].alignement)
18:             Indexes[e] ← index
19:             index ← index + sizeof(Allocas[e])
20:             Allocas.pop(e)
21:         P_Table.append(Indexes)
22:         Empty(Indexes)
23:     return P_Table
```

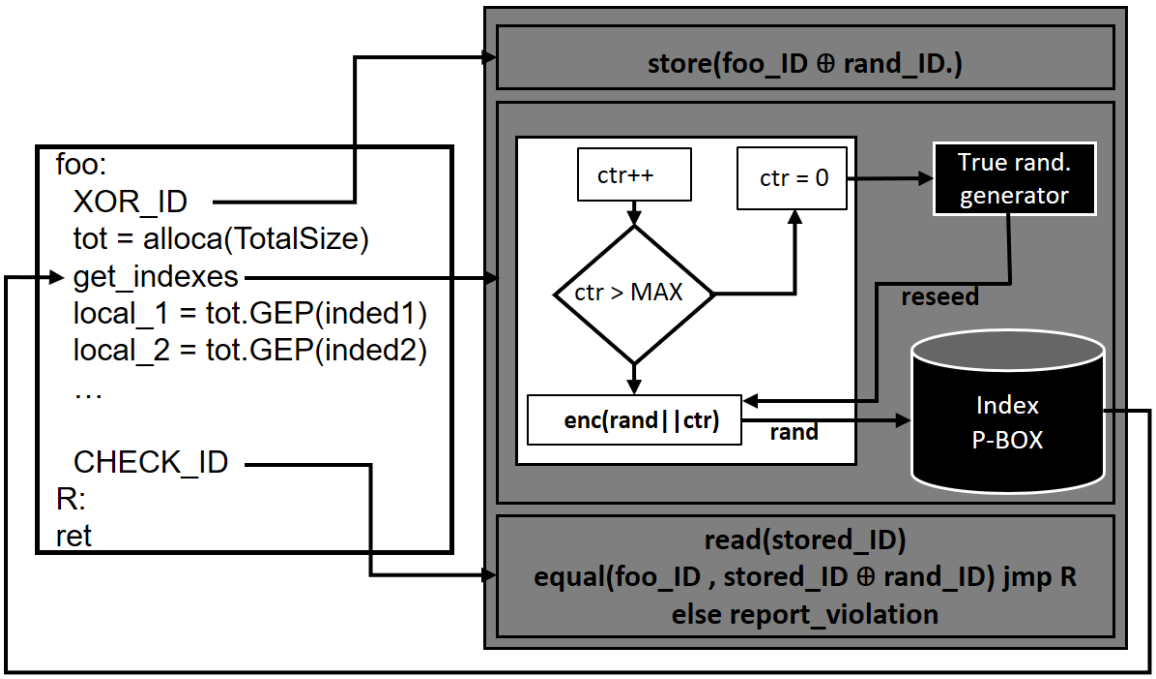


Figure 4.2 Function call and return in a Smokestack. This figure shows an overview of the stack layout randomization steps involved in Smokestack function calls and returns for the `AES` based random number generation schemes.

Generating Random Permutation In this stage, we generate all the possible permutations of stack allocations of each function. To do this, we represent each stack allocation as an index from the start of the stack frame, a size and alignment requirement. We then generate all possible permutations, in lexical order, for all the stack allocations within the function. Algorithm 4.2 shows the pseudo-code of our permutation engine. The permutation engine takes all the stack allocation in a function as an input. At each iteration of the outer loop, it generates a unique permutation of all the allocations within the total allocation. After computing all the iterations, we will have a table, where each row in the table contains a set of indexes for a given random permutation of allocations in the function. We then permute the rows in a table to avoid the lexical correlation between any two consecutive rows in a table. Finally, we store the permutation table of the function, which holds the indexes of each allocation in the total allocation of the its stack frames for every possible order of allocations, in the P-BOX. A P-BOX is shared among all functions with the same stack format. In section 4.3.5, we show how we reduce the associated memory overhead by sharing permutation tables between different functions. Tables in the P-BOX are indexed by a random number, generated at function invocation, to get the indexes of the local variables from the base of the stack frame for that particular permutation.

Runtime Allocation of Randomly Permuted Stack Frames

This phase instruments the program in order to randomize the stack layout of each function call by randomizing the order and alignment of all of local variables. This is achieved by maintaining a single stack frame allocation with a size equal to the total allocation and replacing all stack variable allocations in the stack frame with a slice into this total allocation. Figure 4.2 shows the instrumentation introduced by Smokestack. Upon a function invocation, a random permutation of the local variables is chosen, using a random number to index the table associated with the function in the P-BOX to get a row of indexes. Then, allocations in the stack frame are assigned to their respective slices within the total allocation based on the indexes in the row of indexes. This will ensure that the absolute address and the relative distance to a stack resident object, which can be used in a DOP gadget, is unpredictable for each invocation of a function.

Random Number Generation We considered various random number generation schemes at the beginning of each function to choose a random permutation of local variables. We considered any form of pseudo-random number generation, in which the algorithms state is in memory, as unsafe, since a powerful DOP-oriented attacker could certainly read and manipulate the state of a memory-based pseudo-random generator.

- *Generating a true random number at the entry of each function.* For this scheme, we tested a true random number generator (*i.e.*, `/dev/random`) and `rdrand`, the on-chip hardware random number generator on Intel processors. As `/dev/random` stalls when the system's internal entropy pool is exhausted, we tested only `rdrand` on our prototype implementation.

- *Generating cryptographically secure pseudo-random number.* For this scheme, we generate encryption keys using true random number. This is accomplished by maintaining a universal call counter to track the number of function calls before generating a new true random number to guarantee strong source of entropy. At the entry point of a function we generate a pseudo random number using AES counter mode encryption by using the last generated random number as an initial value and the call counter as a counter. We used the Intel's AES-NI extensions [82] to accelerate our random number generation. On our prototype implementation, we tested by varying the rounds of the AES encryption to see the trade-off between security and performance.

Our instrumentation defers randomization of allocations whose size cannot be determined at compile to runtime by adding a random sized padding on top of the static total

allocation. Variable length arrays (VLA), which are supported in the C99 standard, are one such example. We randomize the layout of stack frames with VLAs at runtime by adding a random sized dummy `alloca` before each VLA in a stack frame. This randomization guarantees that both the absolute address of the VLA and its relative distance from other stack resident objects is indeterministic.

Protecting Smokestack Defenses

Finally, the instrumentation phase adds checks to detect attacks that bypass our instrumentation, for example, by jumping into the middle of a function. To achieve this the instrumentation phase adds a unique load-time identifier for each function, which is XOR'ed with a random key at the prologue of the function. At the epilogue, it is XOR'ed again with the random key and checked against the function identifier. These checks, together with the the stack runtime stack layout randomization, can be a second line of defense for control-flow attacks.

4.3.5 Performance and Memory Size Optimizations

To reduce the performance overhead and the memory footprint of our instrumentation, we applied the following optimizations to Smokestack:

- *P-BOX size of power of 2.* This optimization rounds up the size of P-BOX from *n!* to the next power of 2. This is achieved by wrapping around indexes *n! to next-power-of-2*. It allows the replacement of a modulo operation in our instrumentation with a much faster left shift operation.

- *Rearranging Stack Allocations* This optimization rearranges stack allocations of functions to use existing P-BOX entry tables if a match table is already in the P-Box. For example, function `f1` with local variables `int, double` can share a P-BOX entry with function `f2` with local variables `double, int`. This optimization reduces the associated memory overhead and doesn't have any effect on the performance as the actual order of the variables in the resulting binary is determined by subsequent phases of the compilation.

- *Rounding up Allocations.* This optimization reduces the memory usage of P-BOXs by sharing a table for functions having stack frames that differ only by one primitive allocation. For example, functions `f1(double, double, int)` and `f2(double,`

`double)` can share a P-BOX table at the expense of extra padding in the stack frame of `f2`.This optimization takes advantage of the fact that the least significant indexes within a permutation entry of a smaller table is same as permutation of a bigger size in lexical order of permutation. This optimization also improves performance for frequently called functions.

4.4 Implementation

We implemented Smokestack on top of the LLVM 3.9 compilation framework [83], modifying the LLVM libraries and the compiler-rt runtime. Our analysis and instrumentation passes operate on LLVM intermediate representation (IR), which is generated from source files using the LLVM `clang` front-end.

4.4.1 Analysis passes

We implemented P-Box generation in several LLVM passes. The first function pass gathers all stack allocation for all functions that have an on-stack memory object. Then, a module pass uses the meta-data generated by the function pass to generate a P-BOX table for each function, considering the alignment requirements and the optimization discussed in Section 4.3.5. The final analysis pass generates the P-BOX for all the unique P-Box entry tables.

Alignment requirements. For primitive types, their alignment requirement can be extracted as part of the IR instruction. For aggregate and user defined types, we have to consider both element alignment requirements and aggregate alignment requirements. An element could be a primitive type whose alignment requirement can be extracted easily or an aggregate type, in which case the process is recursive. Aggregate alignment requirements, on the other hand, depends on the alignment requirement of the largest element in the aggregate type.

4.4.2 Instrumentation passes

The instrumentation pass inserts an allocation with the size of the total allocation at the beginning of the function and inserts a call to a random number generator `inline` library function. Then, it replaces all the `alloca`'s in the function with `getelementptr` whose index is decided by the generated random number. Finally the instrumentation pass inserts checks to detect attacks that bypass the stack allocation instrumentation.

4.5 Evaluation

This section presents the detailed performance and security evaluation of Smokestack. We ran our experiments on an Intel Xeon D-1541 processor, running Ubuntu 16.04 Linux with 32GB memory.

4.5.1 Performance Evaluation

We evaluated the performance overhead of Smokestack using SPEC 2006 benchmarks and I/O bound real-world applications, *e.g.*, ProFTPD, Wireshark. We ran four experiments, which varied in implementation of random number generation. *pseudo* utilizes a memory-based pseudo-random number generator. This is only included as a performance baseline, as it is considered completely unsafe by our threat model (since the attacker can expect the state of the generator at any time). The *AES-1* and *AES-10* experiments use the Intel AES-NI instructions to repeatedly re-encrypt a true-random seed, with the former experiment only running one AES round and the latter running all 10 required by the specification. Finally, *RDRAND* uses the Intel RDRAND instruction to get a true-random number for use by the stack layout permutation code.

Figure 4.3 shows the performance overhead of Smokestack for the SPEC 2006 benchmarks. Our measurements show that the performance is dependent on the way we generate a random number. For unsafe pseudo-random number generation, the normalized performance varies from a speedup of 2.6% to a slowdown 7.2%, averaging to 0.3% slowdown over the SPEC2006 benchmarks. Using a cryptographically secure pseudo random generation (AES-128 10 rounds), the overhead spans from 0.6% up to 20% and averaging 8.7%. To assess the overhead vs. security trade-off, we also examined the performance of a less secure pseudo-random number generation (AES-128 1 round), which has an average slowdown of 2.2%. For RDRAND-based true-random number generation, there was greater slowdown due to the bandwidth limitations of the true random number generator. This experiment experienced an overall slowdown of nearly 19%.

To examine the source of performance gain on our benchmarks, we ran the Oprofile [84] tool with our SPEC 2006 experiments, which clearly show the variation on the *RESOURCE_STALLS* parameter, depending on the benchmarks. Our analysis illustrates that this result is due to instruction scheduling and register pressure by Smokestack. On some benchmarks, Smokestack increases register pressure and consumes load delay slots during the CPU scheduling. The register pressure improved performance on benchmarks where registers are underutilized, and degrades performance if registers were already fully utilized.

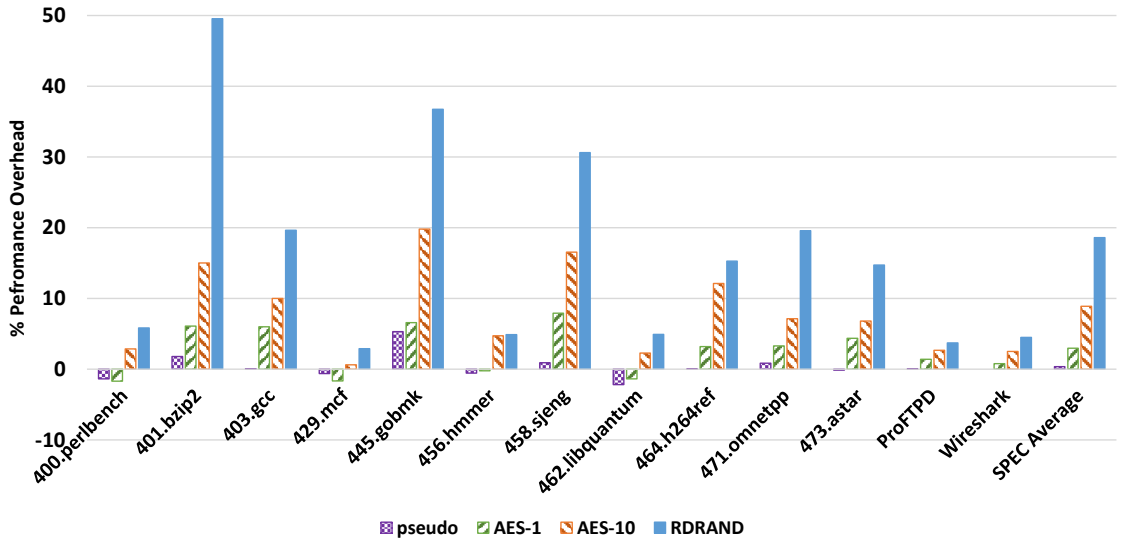


Figure 4.3 Percentage performance overhead of Smokestack. This figure shows the percentage runtime overhead of Smokestack on SPEC2006 benchmarks. The experiments varied based on the method by which we generated random numbers. `RDARAND` shows the use of `rdrand`, the on-chip random number generator on Intel processors. The others show use a cryptographically secure pseudo-random number, AES-128 counter mode with 10 rounds (`AES-10`) and less secure variant with 1 round (`AES-1`). Finally, `pseudo` shows the overhead of using an insecure memory-based pseudo-random number generator.

On I/O bound applications, we used for our performance evaluation, ProFTPD and Wireshark. Smokestack incurs negligible overhead, with the worst case performance overhead of 5%.

4.5.2 Memory Overhead

We evaluated the memory overhead of Smokestack by measuring the maximum resident set size (`ru_maxrss`) while running SPEC 2006 benchmarks. Figure 4.4 shows the results of these experiments. It's interesting to note that benchmarks with higher memory overhead, like *perlbench* and *h264ref*, have lesser performance overheads. This is due to the fact that the source of the memory overhead is the addition of the index P_BOX in the read-only data section, which doesn't strongly affect the I-cache miss rate.

4.5.3 Security Analysis

In this section, we assess effectiveness of Smokestack in protecting against DOP attacks. To this end, we first analyze the security vs. data rate of the sources of randomness we used for

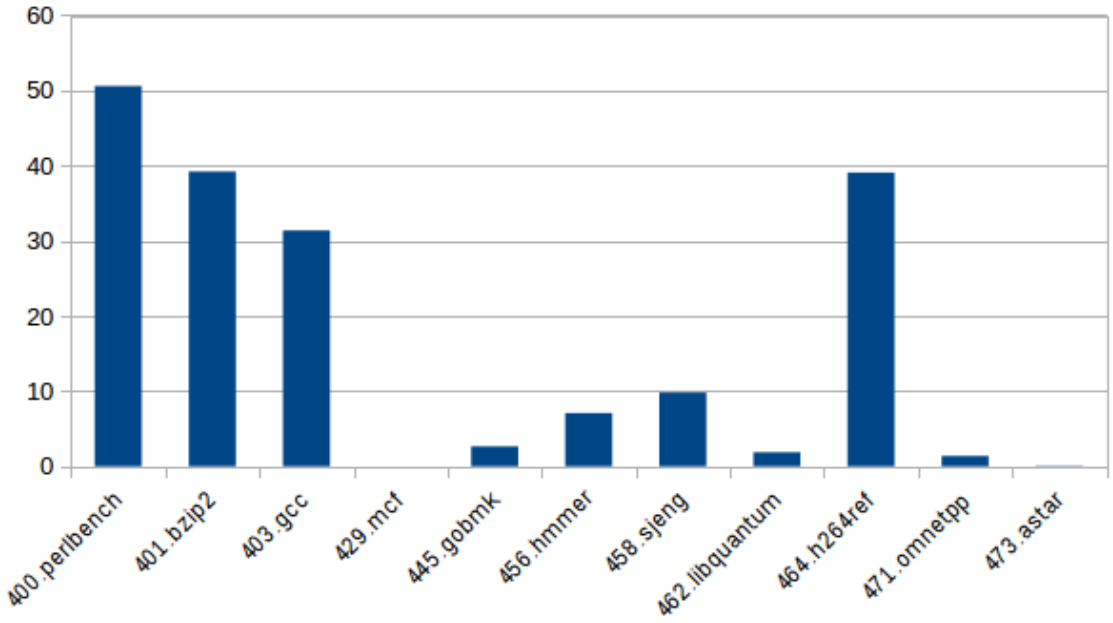


Figure 4.4 Percentage memory overhead of Smokestack. This figure shows the percentage increase in maximum resident set size of Smokestack on SPEC2006 benchmarks.

Table 4.1 Smokestack source of randomness. shows the rate at which random values can be generated by the random generator schemes we tested for our prototype implementation.

source	Security	Rate (cycles/Invocation)
`pseudo`	None	3.4
`AES-1`	Low	19.2
`AES-10`	High	92.8
`RDRAND`	High	265.6

our prototype implementation. We then evaluate Smokestack's effectiveness in protecting DOP attacks in both synthetic benchmarks and real-world applications.

Source of randomness. We performed tests to examine the rate at which we can generate random numbers. Table 4.1 shows the rate at which we can generate random numbers, using random generation schemes with varying security guarantees, back to back on our test machine. While *pseudo* is fastest, it also offers no protection. *RDRAND*, in contrast, provides true random values for each invocation, but at a great delay. The *AES* based random number generators provide a convenient trade-off between security and performance, with overall quite good performance.

Penetration testing with synthetic benchmarks. We synthesized two types of DOP attacks that exploit buffer overflow vulnerabilities to control local variables used as DOP

gadgets and also a loop counter used as gadget dispatchers. The first set of attacks use a stack based buffer overflow vulnerability to corrupt variables in the stack to perform the attack. And, the second set of attack overflow a buffer in data segment or heap to overwrite local variables in the stack. We also consider two types of overflows, direct and indirect (overflows a buffer until a pointer is corrupted, and then uses an assignment through the corrupted pointer to overwrite the target pointer) —an approach followed by the RIPE [85] control-flow attack benchmark suite.

Smokestack is able to prevent all the attacks by breaking the DOP gadgets and gadget dispatchers. All of the direct overflow attacks based on any buffer were stopped. Also, any indirect overflow attacks based on buffers in the data segment or heap corrupted unintended locations in the stack, including padding and function IDs, were stopped. All of the indirect overflows attacks failed on the first step, as they overwrite a different address than the intended pointer that is used to write to the target pointer.

Real Vulnerabilities. In our final set of security analyses, we tested Smokestack's ability to protect against attacks that exploit real vulnerabilities including our own proof-of-concept DOP attack on *librelp* logging library. The following reported DOP attacks were considered for our analysis:

Wireshark network protocol analyzer prior to version 1.8 has a stack-based buffer overflow vulnerability (CVE-2014-2299 [86]) in mpeg reading function *cf_read_frame_r()*. This vulnerable function is called from *packet_list_dissect_and_cache _record ()* to copy user specifier mpeg frame data to a fixed sized buffer *pd*. Hu et'al. [75] exploited this vulnerability by sending a maliciously crafted trace file that contains a frame larger than the buffer size (0xffff). Their DOP exploit repeatedly overflows the buffer to overwrite variables *col, cinfo*, and parameter *packet_list* in the same function, *i.e., packet_list_dissect_and_cache_record ()*, and the loop condition *cell_list* in parent function, *gtk_tree_view_column_cell_set _cell_data()*, with malicious input. *col, cinfo and packet_list* are used as DOP gadget operands and *packet_list* used for stitching together gadgets in the subsequent calls to the function *packet_list_change_record()*, which contains all the DOP gadgets. We run this attack on a Smokestack-hardened version of the vulnerable Wireshark version. Smokestack stopped this attack by detecting the violations when the overflow corrupted unintended critical data, like *Smokestack function identifier*, as Smokestack changes the index of *pd* in the stack frame for every call of the function.

ProFTPD. has a stack-based buffer overflow vulnerability (CVE-2006-5815) due to the use of *sstrncpy(dst,src,negative argument)* in the *sreplace()* function [87]. Hu et al. has

demonstrated DOP attacks[1], which extract private keys bypassing ASLR, simulate remotely controlled network bot and alter memory permissions by exploiting this vulnerability.

Extracting private keys bypassing ASLR: ProFTPD stores its OpenSSL private key in a buffer which has a chain of 8 pointers pointing to it with only the base pointer not randomized. A successful attacks requires using memory disclosure to de-randomize 7 global pointers. Hu *et al.* used a DOP attack composed of 24 DOP gadget chains (this requires corrupting operands of virtual operations consisting *MOV, ADD* and *LOAD* for 24 iterations) to successfully extract the OpenSSL private key bypassing ASLR. This attack was demonstrated to bypass TASR [88], which does fine-grained re-randomization of code on every output system call.
The other two DOP attacks on ProtFTPD use *sreplace()* to corrupt relocation metadata in the *link_map* structure. This is then used by *dlopen()*, which is invoked in its PAM module to dynamically load libraries, to process the corrupted relocation metadata. The two end-to-end exploits used this to simulates a bot that repeatedly responds to network commands and alters memory permissions to bypass defenses, including $w\hat{x}$, *.rodata* and CFI defenses that use read-only legitimate address tables.

All these attacks repeatedly use a memory corruption vulnerability in *sreplace()* to chain together virtual instructions used in the DOP attack by repeatedly overwriting a loop counter, which is used as a DOP dispatcher. We were able to detect all the attacks on the Smokestack-hardened version of the affected version of ProFTPD. Smokestack was able to stop this attack by randomizing the address of the loop counter used to stitch the DOP gadgets together, hence breaking the gadget chain.

4.6 Related work

Several memory corruption attack mitigation techniques have been proposed. These protections can generally be categorized in to two classes. The first class is enforcement-based protections that perform explicit checks based on predefined policies. These techniques vary from protection, which guarantee full spatial memory safety, such as Softbound [27], to attacks that target particular type of exploit, such as CFI [32], that protects against control flow hijacking exploits.

The other class is randomization-based protections, in which a critical asset used by an attacker for a runtime attack is randomized after it is acquired and before used in a payload. Randomization-based solutions are more efficient, incurring very low to no overhead, than

[1] https://huhong-nus.github.io/advanced-DOP/

enforcement-based solutions. Address space layout randomization (ASLR) [5] is a widely deployed randomization-based technique. It randomizes the base of sections of a program, such as code, stack, heap and shared libraries in its address space during load time. However, ASLR is shown to be ineffective in the presence of even a single memory leak [89] or brute-force attacks [49]. Successive improvements to randomization-based techniques were proposed to increase entropy by decreasing the granularity of the randomization to function level [37], basic-block level [39], and instruction level[40]. However, subsequent works [21][90] have shown that compile-time and load-time fine-grained randomizations can be bypassed by runtime attacks that dynamically generate their payloads. Recently, periodic re-randomization [91] has been shown to be effective in stopping runtime attacks. But it has only been validated for code pointer protection to thwart control-flow attacks that are resilient to static randomizations.

With the ultimate goal of taking control of the program, control-flow hijacking is usually the easiest and the primary way of exploiting memory corruption attacks. To address control-flow hijacking attacks, a wave of mitigation techniques has been proposed. The leading approach is control flow integrity (CFI) [32], which is based on constructing the program's CFG prior to its execution and validating at runtime whether the execution path follows a valid edge in the CFG. With the advent of low overhead CFI techniques to protect corruption of control data, non-control data attacks have received significant attention by attackers.

Several works propose to mitigate attacks based on non-control-data, including enforcement, randomization and language-based approaches.

Data-Flow Integrity (DFI) [34] statically performs reaching definition set analysis of instructions. DFI then instruments read access instructions to ensure that the last instruction that last wrote to the location is within the reaching definition set of the instruction. Even though DFI is capable of mitigating DOP attacks, a complete DFI protection incurs a very high overhead (50 - 100% on SPEC 2000 benchmarks).

PointGuard [42] proposed encrypting all pointers when they are stored in memory and decrypting them just before they are loaded into registers. However, it uses a single key to XOR all pointers, hence a single leak on known encrypted pointer from memory can be used to recover the key. Data Space Randomization (DSR) [43] tries to solve the shortcomings of PointGuard by using a different key for all variables. However, even that has been revealed as ineffective in the face of memory leaks [89].

Giuffrida et al.[80] proposed an ASR technique that periodically performs live re-randomization of program modules. Unfortunately, their re-randomization technique induces significant runtime overhead when applied when the re-randomization period is small (e.g.,

50% performance overhead for re-randomization period 1 sec). In addition, their proposed technique is tailored to microkernels, relying on hardware-isolation and runtime error recovery. Moreover, their stack randomization is static randomization and randomized padding that can be bypassed by the techniques we presented in section 4.2.3.
YARRA [19] provides a C language extension for validating sensitive pointers pointing to a critical data, such as secret keys, in the program as annotated by the programmer. It does these by using page protection to lock its protected data when running unsafe procedures. YARRA offers a security guarantee for non-control data attacks against annotated data. However, it incurs a very high overhead for protecting the whole program (*e.g.*, 6x overhead on gzip).

HardScope [92] ensures an intra-program memory compartmentalization by enforcing compile-time discovered variable scope constraints at run-time. HardScope instruments memory accesses at compile-time to check that the memory address requested is within the allowed memory areas. HardScope was demonstrated by extending the RISC-V instruction set with six new instructions. Even though Hardscope has a low overhead, it requires a hardware support. In addition, it is still susceptible to DOP attacks that share access to the same global data structures or have a data flow reaching a global data structure.

4.7 Chapter Summary

With widespread adoption of control-flow hijacking attack mitigations, DOP attacks have become an increasingly popular source of attacks against systems. While randomization-based mitigations are gaining popularity due to their efficiency, we found their protections to be ineffective at stopping DOP attacks. We also found previously proposed stack randomization efforts to be ineffective at stopping DOP attacks. Smokestack gains additional power at stopping DOP attacks by randomizing stack frames for functions each time they are called. In addition, we leverage true-random stack layout permutation that is resistant to memory disclosure attacks, forcing the attacker to reverse engineer a function frame in the same invocation that it is attacked. Our proof-of-concept implementation in LLVM demonstrates that the approach can effectively stop DOP attacks with only minimal slowdown in program execution.

Chapter 5
Runtime Heap Layout Randomization

5.1 Introduction

Programs written in C and C++ are inherently vulnerable to memory bugs, including buffer overflows and use-after-free. These memory bugs in optimistic scenarios cause program crashes or degrade the performance of the program. In worst case scenarios, they can be exploited to perform security attacks, such as remote code execution to steal sensitive information or denial-of-service (DoS) to impact availability of services. Despite significant advances in techniques to find memory errors during the testing stage of programs (*i.e.*, before they escape to production systems) [93] [29][94], it has been apparent that it is very difficult to eliminate all memory errors through testing. This is partly due to the ever increasing code base in modern systems. Hence, proactive preventive measures against the exploitation of memory errors provide a viable approach and continue to be widely deployed in modern systems.

Early system level exploits of memory corruption vulnerability, such as stack smashing, relied on using memory errors to corrupt a code pointer on the stack to hijack the control flow of the program. These early attacks were mitigated by stack based protections, such as stack canaries. Attackers then added heap based exploits to their playbook in response to the wide spread adoption of stack resident code pointer protections. Heap based attacks commonly corrupt either the allocator metadata or the allocated data itself using memory access errors. Based on the allocator's management of metadata, control flow hijacking attacks are possible through corrupting the metadata. For instance, by corrupting the header on free-list based allocators, such as Windows and DLmalloc allocators that store the free-list metadata adjacent to heap resident buffer, it is possible to jump to a shell code when the corrupted free chunk is reallocated [95].

Even though secure allocators, such as OpenBSD's allocator, provide probabilistic guarantees against these attacks that require the knowledge of the absolute address of a given allocation, they do not provide strong security guarantees for attacks that only utilize

relative distance between allocations. In addition, even though these allocators ensure their allocation and reuse is randomized, an allocated buffer stays at the same location until it is freed, which can last as long as the entire runtime of the program. However, the relative distance can be disclosed considering the abundance of memory leak attacks and can be exploited using non-linear buffer overwrite techniques.

In this work, we make an observation that it is essential to randomize heap allocations throughout their life time to mitigate attacks that can discover allocations through memory disclosure. To mitigate attacks that utilize only relative distance between allocations, we relocate chunks of memory used by the heap allocators on output system call granularity. We utilize multi-variant execution (MVX) with carefully diversified variants to keep track of pointers. Our design leverages invariants across the variants to identify pointer values from non-pointer data by comparing the variants' memory.

Based on this observation, we propose `HeapRand`, which builds on top of a Dune [96]. Dune utilizes hardware virtualization features in modern systems (Intel-VTx) to accelerate intra-process isolation and provide a safe and direct access to virtualization hardware features for user space processes. Once a process enters in Dune mode, it uses hypercalls to invoke system calls, using the same principle devirtualization uses for file input output operations. Dune, more importantly, doesn't suffer from the same performance degradation as devirtualization due to blocking, as it runs processes in their own separate virtual space.

In sum, in this work we make several key contribution towards defensive research against heap based code-reuse attacks. First, we present a heap layout randomization technique called `HeapRand`. In addition, We introduce a metadata tracking technique without explicit tagging of memory addresses, by running multiple variants, with each having a carefully designed program diversification to identify pointer values from other memory regions. Moreover, we present the evaluation of `HeapRand` on SPEC CINT 2006 benchmarks.

The remainder of this chapter is organized as follows. Section 5.2 presents a detailed background on heap based memory corruption exploits and requirements for `HeapRand`. Section 5.3 details the assumed threat model and the design of `HeapRand`. Implementation details are presented in Section 5.4. Section 5.5 presents the evaluation of `HeapRand` . And finally, section 5.7 concludes the chapter.

5.2 Background

Before delving into the design and implementation of `HeapRand`, in this section, we present the background of the problem and technique used to develop our solution. We

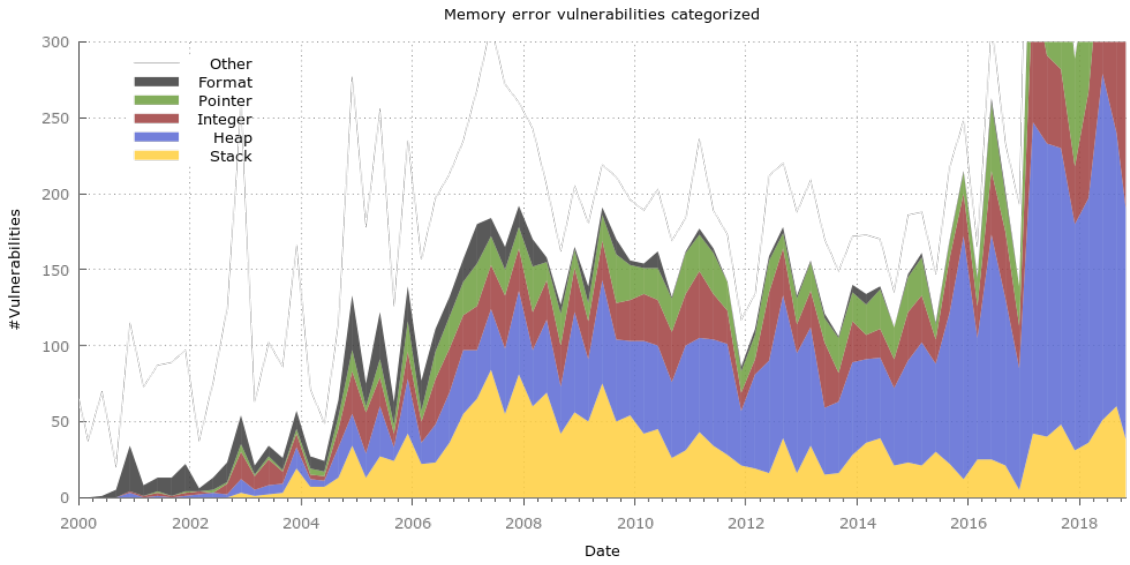


Figure 5.1 Trends in memory error vulnerabilities [97]

specifically discuss heap based memory vulnerabilities, the state-of-the-art in securing heap allocations and the hardware assisted virtualization technology we based our work on.

5.2.1 Heap Based Memory Vulnerabilities

Figure 5.1 shows the number of heap-based errors, such as use-after-free, heap overread, and heap overflow, discovered in the past two decades (extracted from the CVE data feed) [1]. This trend depicts that the stack based attack was a more prevalent attack vector in earlier days of exploitation whereas, more recently, the heap has gotten attackers attention as a result of widespread adoption of stack based protection, such as stack canaries.

Heap Overflows. A heap overflow/underflow occurs when a program writes outside of the boundary of an allocated object. Like stack based overflows, the overwrite could be a linear overwrite (*i.e.*, sequentially overwriting all memory addresses starting from the buffer being overflown) or offset (non-linear) overwrite(*i.e.*, skips intermediate addresses and corrupts only the intended memory addresses). Heap overflows can be utilized for malicious purposes, such as privilege escalation, code execution and program crashes. Offset based overflows in particular are stealthier, as they can be used to overwrite the target address without corrupting the address in between; hence, they are able to bypass protections, such as heap canaries.

[1] https://nvd.nist.gov/vuln/data-feeds#CVE_FEED

Heap Over-reads. Heap over-read happens when a program reads memory that was not intended to be accessed through a heap allocated object by overrunning its boundary. Heap under-reads, in which memory locations prior to the heap allocated object are accessed, is generally categorized under this category. Heap over-reads can cause erratic program behavior such as memory access violation (*i.e.*, if the access reached an unmapped region), memory disclosure or denial-of-service attacks.

Uninitialized Reads. Uninitialized reads happen when a program reads from a newly allocated heap object. Similar to heap over-reads, it can cause a memory disclosure, as the newly allocated object may contain data from previously freed allocations.

Use-after-free. Use-after-free happens when a program prematurely frees a heap allocated object and continues using it. *Double frees* are one type of use-after-free. in which the program frees a previously deallocated object. Depending on an allocator's handling of freed objects, use-after-free vulnerabilities can have detrimental security consequences, including memory leaks that may cause denial-of service, code execution and corruption of the allocator's metadata.

Heap Based DOP Attacks. A recent attack demonstrated a heap based DOP attack against GStreamer[2] decoder for FLIC file format [98]. GStreamer has an non-linear heap buffer overflow vulnerability in its decode function. It enables to overflow a JPEG buffer to overwrite an arbitrary target skipping over the intermediate data. The attack uses this vulnerability to repeatedly corrupt heap resident pointers that are used in a chain of dereference, addition and assignment DOP gadgets. The DOP attack is used to de-randomize an ASLR randomized program to obtain a pointer to a function, which is then used for code execution in later stages of the attack.

5.2.2 Heap Allocators

Existing heap allocators can be categorized in two major categories, namely: Sequential/free-list based allocators and BIBOP (Big-Bag of Pages) allocators [95].

Sequential/Free-list allocators These allocators maintain a linked list to keep track of freed allocations, where the lists are grouped by allocation size. Allocated objects maintain allocation metadata, which includes the allocation size, the status of the allocation and size

[2]https://gstreamer.freedesktop.org

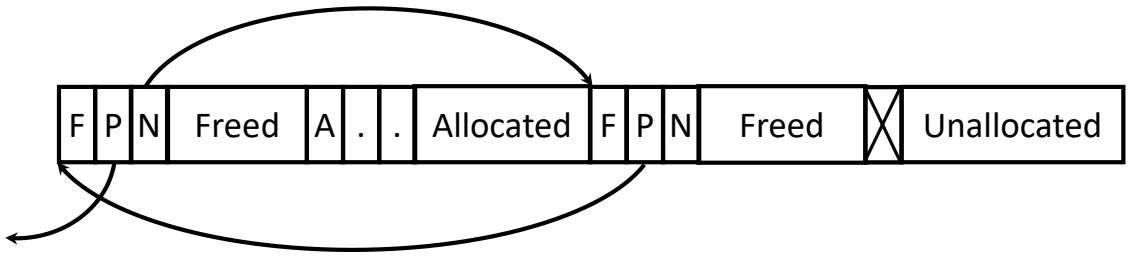


Figure 5.2 Sequential Allocators shows a fragment of memory allocated with sequential/free-list based allocators. Memory objects are prepended by headers which store allocation metadata, including status of the allocation, size and pointer to previous and next freed allocations.

of previous chunk (if allocated) in the allocation header, which is stored prepending the allocated object. Storing the metadata alongside the allocated object enables the allocator to efficiently place freed allocations to their corresponding free-list size class; as well as efficient coalescing of adjacent free allocations. Typical examples of sequential allocators include Window's and Linux's default allocators. These allocators are vulnerable to overflow based attacks, as the the allocated memory that is used by the program is adjacent to the metadata used by the memory management functions themselves. Buffer overflows or underflows can corrupt the data structures that the memory management functions maintains to track allocated and freed memory regions, which can potentially result in control-flow attacks [95].

Figure 5.2 shows a fragment of memory as used in free-list based allocators. The allocation header precedes the allocation and contains metadata used by the memory management function to keep track of the status of available allocations. Even though storing the allocation metadata inline offers a good performance for memory management functions, metadata corruption can have serious security implications. For instance, control-flow hijacking is possible if an attacker manages to overwrite the next pointer of a freed object metadata to point to an attack payload (e.g., ROP gadget) and the previous pointer to point to function pointer used for memory management (e.g., `_free_hook`, a function pointer used during a call to `free()`). When this freed object is allocated, the freelist is updated by writing the content of next pointer metadata to previous pointer metadata, overwriting the `_free_hook` function pointer now to point to the shell code. Then, subsequent calls to `free()` will end up hijacking the control-flow of the program to execute the attack payload.

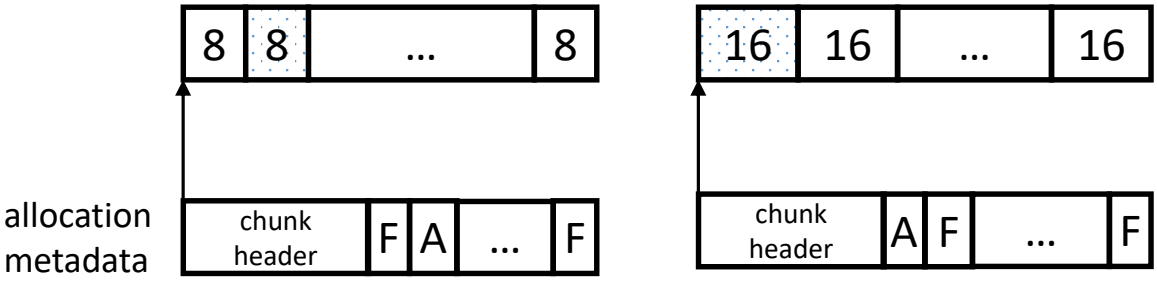


Figure 5.3 BIBOP Allocators illustrates a fragment of memory allocated with BIBOP based allocators. Memory object are stored in a chunk of memory according to their size and the allocator metadata is usually placed in a segregated shadow region to protect from being corrupted with buffer overflow.

BIBOP Allocators These allocators reserve chunks (bags) of memory to allocations with each bag serving a request for a particular allocation size. Bags are usually sized multiple of pages. These allocators generally have less attack surface compared to sequential allocators. as they typically segregate the allocation meta-data from the actual allocations. Hence, BIBOP allocators are not susceptible to meta-data corruption with linear overflow of allocations. Figure 5.3 shows a fragment of memory regions allocated using BIBOP allocators. Typical examples of BIBOP allocators include PHKmalloc and OpenBSD allocators.

5.2.3 Memory Disclosure

Modern systems are replete with randomization based security measures, the main one being Address Space Layout Randomization (ASLR), to protect against a wide range of exploits, such as control flow hijacking and non-control data attacks. The exploits typically require corrupting a code or data pointer to overwrite it with an address of exploit code or data pointer (in case of DOP attacks). ASLR randomizes the address of the code/data pointer overwritten by the exploits and the address of the exploit code, which an attacker is required to know to perform a successful attack. However, as ASLR is implemented as a one time (load-time) randomization technique, its effectiveness relies completely on the confidentiality of the randomized addresses. ASLR has been shown to perform weakly in practice, as attackers can extract a randomized address, which is abundantly available in a program's memory.

Figure 5.4 Overflow to Memory disclosure.

Hence, reliable memory corruption exploits in modern systems rely on memory disclosure to reveal the memory layout or content of various sections of the process/kernel memory. For instance, memory disclosure can be a precursor for a successful attack against an ASLR hardened system. Load-time randomization techniques, such as ASLR[5], ensure that the memory layout of the program will be different for every run. However, it will remain with the one time randomize layout through out the runtime of the program. To circumvent this, there are a number of memory disclosure attacks capable of de-randomizing the layout of fully fine-grained randomized programs at runtime. including JIT-ROP[99] and Blind ROP [22]. JIT-ROP repeatedly exploits an out of bound memory access vulnerability to recursively find memory resident code pointers in order to then extract code pages from a fine-grained randomized program and deploy an ROP payload using just-in-time compilation techniques. Blind-ROP, in contrast, uses the response (whether a process crashed or not) of worker processes of a daemon web server to incrementally obtain the memory layout of the program required to remotely deploy a malicious payload on fully diversified programs.

Moreover, many memory error vulnerabilities, including stack/heap overflow and use-after-free, have proven to be convertible to a memory disclosure attacks [100]. Figure 5.4 shows an example of how a heap based memory disclosure can be converted to a memory disclosure attack. If a string that stages an attacker controlled sized sub-string operation is placed adjacent to an object of interest (such as code pointer), a heap based buffer overflow can be used to overwrite the terminating character of the string. Hence, subsequent sub-string operations performed on the corrupted string will leak contents of adjacent memory locations. Likewise, most other memory corruption vulnerabilities were demonstrated to be amenable to be converted into a memory disclosure attacks, which is a precursor for successful exploits on modern systems equipped with ASLR [100].

5.2.4 Multi-Variant Execution

Multi-Variant Execution (MVX), in which multiple diversified variants' of a program are run to detect variation when malicious input is provided, has been shown as a promising

approach to systems security as it is capable of providing a comprehensive defense against many classes of attacks. Diversified variants of the program are executed in lockstep by a variant monitor/s and synchronized at certain points. Typical synchronization happens during system calls as it is the primary means by which an attacker interacts with the system. The variants are chosen to be semantically equivalent while exhibiting diversity in their memory layout. To exploit an MVX program, an attack needs to exhibit the same behaviour across all the variants. Depending on how they monitor the variants, MVX systems can be categorized as centralized and distributed. The majority of the MVX systems proposed in the literature use a single, centralized monitor component that is placed outside the address spaces of the variants. Additionally, the monitor can run either in user space [101][102][103][104][105] or kernel space [106][107].

The MVX monitor is responsible for synchronizing variants and comparing the variants to detect divergence behavior. Synchronization of variants can be performed at different granularities. A typical approach to synchronize variants is at the granularity of system calls, as it is the primary means the program interacts with the outside environment. In addition, MVX systems should present all variants running concurrently as a single program. Hence, the monitor should synchronize the diversified variants and pose as a single program to the user. The MVX system accomplishes this by making system calls that should only be performed once to be executed by a leader variant and distributing the result to all the other variants. This includes I/O operation in which a single input is duplicated to serve each variant and outputs from all variants are compared; and a single output operation is performed by the monitor in order to create the impression that the user is interacting with a single instance of the program.

The synchronization point is a defining character for MVX systems. MVX systems that focus on reliability and availability can have a wide synchronization window, where variants can run out of sync most of the time. MVX systems that allow variants to run asynchronously need to have a shared ring buffer where the leader variant can store the results of the system call, which then are consumed by all follower variants. On the other hand, MVX systems that are security focused have a low synchronization window, which is based on the type of security guarantee. For example, MVX systems that target memory disclosure need to synchronize at output system calls.

Some recent MVX systems [108][109][110] utilize distributed monitoring of variants, wherein monitors can be placed at each variant's address space and communicates through a shared memory region. The shared memory region usually is composed of a ring buffer that is optimized to hold system call arguments and their return values. In addition, some recent MVX systems [106][108] offer a capability to setup extra ring buffers that can be utilized

for additional divergence detection below the system call interface.

The use of non-overlapping address spaces across variants can effectively stop all attacks that rely on absolute address in the address space of the program, such as return-into-libc and ROP attacks. However, it doesn't give any guarantees against attacks that rely on relative distance between allocations, including most non-control data attacks. Prior works have proposed different techniques to overcome this. MvArmor [109] ensures non overlapping offsets across variants by using a compact allocator in the leader variant and maintaining the distance between allocations in the other variants to be larger than the entire heap in the leader, using a sparse allocator to stop relative distance based attacks.

Variant Generation MVX systems generate and run multiple randomized variants of a program to detect variations in the execution of the program when under attack. The randomization introduced in the variants has to preserve the semantics of the program, all randomized variants must give the same output, and yet must also exhibit difference when the program is under attack. On the one hand, having too much divergence in the variants may result in a scenario where it is impossible to differentiate divergence introduced by the MVX system design from a divergence introduced by an attack. On the other hand, having too few divergence among the variants may result in an attack having the same effect across variants, making the MVX system unable to detect attacks.

To mitigate attacks that rely on the relative distance between allocation `HeapRand` randomizes all allocations during output system calls. To achieve this, the monitor keeps track of all allocated pages in the program. For all the variants, we use a randomized allocator with different randomization seeds. During the randomization, the monitor remaps all allocated pages in the program.

5.3 Runtime Heap Layout Randomization

5.3.1 Threat Model

`HeapRand` provides protection against memory corruption attacks based on exploitation of linear/non-linear spatial memory bugs as well as temporal memory bugs on heap allocations. To this end, we assume a strong threat model where an attacker has an access to interact with the target program arbitrary number of times to perform linear/non-linear buffer over-reads (e.g., to disclose the memory layout of the program) and perform linear/non-linear overwrites (e.g., to corrupt code/data pointers). We also assume that the attacker is capable

of performing repeated attacks as well as allocate and free objects at will.

5.3.2 Using MVX for Heap Layout Randomization

In this section, we present the design of `HeapRand` based on MVX system. MVX systems run multiple variants of a given program, providing the same inputs for all the variants while monitoring for variations in their behaviors. `HeapRand` is based on the observation that when multiple variants of a program run on MVX, at blocking system calls, their memory content should only differ for pointer values. To achieve this, `HeapRand` utilizes a Dune [96] based MVX system to randomize the address of chunks used in BIBOP stype allocators.

5.3.3 Process Virtualization with Dune

Dune utilizes Intel's VT-x technology to provide user space programs to safely use privileged CPU features, such as ring protection, tagged TLB and management of their own page tables. Hardware-based paging afforded by Dune has a significant performance improvement over software-based paging [111]. Dune exposed guest page table that maps guest-virtual addresses to their corresponding guest-physical addresses for user-level processes. The Extended Page Table (EPT), which is managed by the underlying kernel, performs the guest-physical to host-physical address translations. Figure 5.6 shows the address translation of Dune processes.

`HeapRand` leverages Dune's hardware based secure in-process monitoring features, adopting the approach followed by MvArmor [109]. MvArmor uses Dune's process virtualization feature to implement its monitor. This helps to avoid the expensive context switches that are observed with monitors implemented in user space. Figure 5.5 shows the system call control flow for `HeapRand` using Dune.

5.3.4 Randomizing Relative Distance between Allocations

`HeapRand` aims to re-randomize chunks based on BIBOP allocations by mapping a new shadow guest-virtual page to the same guest-physical page of each chunk and updating all pointers to allocations in the chunk. The two possible design options to achieve this are either maintaining the randomization metadata in the kernel or storing it into a separate user space process that traces the running process. Keeping the metadata in the kernel has the advantage of a direct access to privileged features, such as page tables, direct page accesses

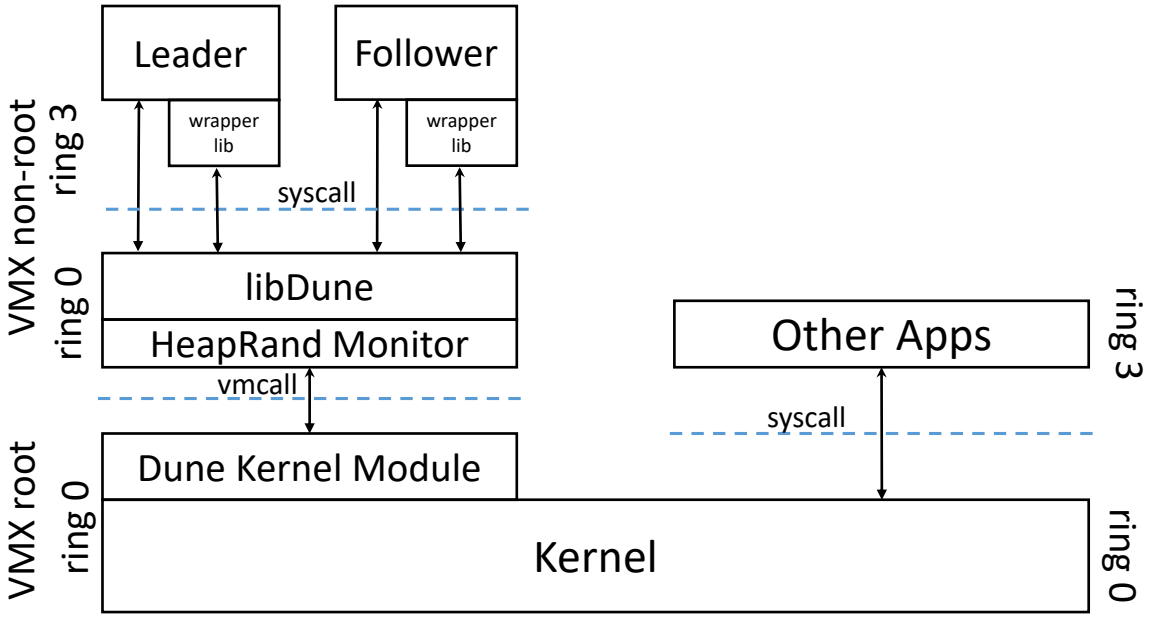


Figure 5.5 Syscall control-flow `HeapRand` using Dune

and other features that facilitate the randomization. However, it has the a disadvantage of increasing the possible attack surface, as a bug in the implementation of the kernel module can affect the entire system. Maintaining it in a separate user space process, in contrast, has the benefits of limited attack surface if the system is compromised and requires minimal change to be used in existing systems. However, it requires frequent context switching between the tracer process and the process being randomized. Dune based MVX monitors have the benefit of accessing the privileged features required for our randomization while running in the same address space as the process.

`HeapRand` runs two variants of a process with each variant using a randomly seeded BIBOP allocator. BIBOP allocators, such as OpenBSD [112], randomly choose an allocations from a set of chunks available for a particular size at runtime. In addition to the randomized allocation, these allocators add a random timeout to the reuse of freed allocations. These properties guarantee that there is no inferred relative order among allocations based on the temporal locality of the memory management function invocations. This is contrary to sequential allocators, which provide no such guarantees. `HeapRand` works to re-randomize the address of chunks used for allocations by randomizing the virtual addresses of pages used by each chunk in the BIBOP allocators. Due to this, it works well with BIBOP allocators, which maintain the size of the bag used in allocations for each allocation class aligned at page boundaries and have a size of a page. OpenBSD is one such example.

`HeapRand` also needs to update the allocation metadata. Luckily, BIBOP allocators maintain their allocation metadata in a segregated region of memory. The allocation metadata is located by obtaining the start address of the page for a given address to find its corresponding metadata. Some allocators, such as Freeguard, maintain the allocation metadata at a randomized offset from the base of the heap which is determined during the initialization of the allocator. Others, including OpenBSD, Dieharder and Guarder, maintain a hash table to keep the mapping from pages used to store bags for a particular allocation size class to its corresponding allocation metadata.

`HeapRand` re-randomizes the relative distance between allocation pages by mapping another virtual page to the same physical page as the original allocation. This is achieved by manipulating the user page table entry for the guest-virtual address of the new chunk to point to the guest-physical address of the old chunk. Once all the pointers to allocation in the old virtual page are updated to point the new virtual page, the page table entry for the old page is invalidated, making subsequent accesses to virtual addresses invalid. As memory disclosure to leak addresses of multiple allocations will involve several output system calls, `HeapRand` can guarantee that any leaking of a memory address to reverse the randomization provided by the allocators will be futile.

Tracking Pointers

To re-randomize heap allocation, we need to track pointers to heap allocation in order to update them upon randomization. There are various options to keep track of pointers.

- Trampoline: Using a trampoline requires replacing all heap allocation pointers in the program with a pointer to an entry in a trampoline table (which stores the actual pointers to the allocation). It also requires memory management functions, like `malloc` and `free` to accommodate these.

- Tagged Memory: This requires a specialized hardware and compiler to maintain the tags of various memory regions.

In our design, we instead rely on variations among diversified execution of the same program to identify pointer values from non-pointer data. We make the observation that by running multiple replicas of a program, at synchronizing points we can identify pointers based on their variation in their values across variants. This property enables us to keep track of heap pointers in all regions of memory without explicitly storing a tag meta-data. As deviation in the values are detected at system calls that synchronize variants, the state of

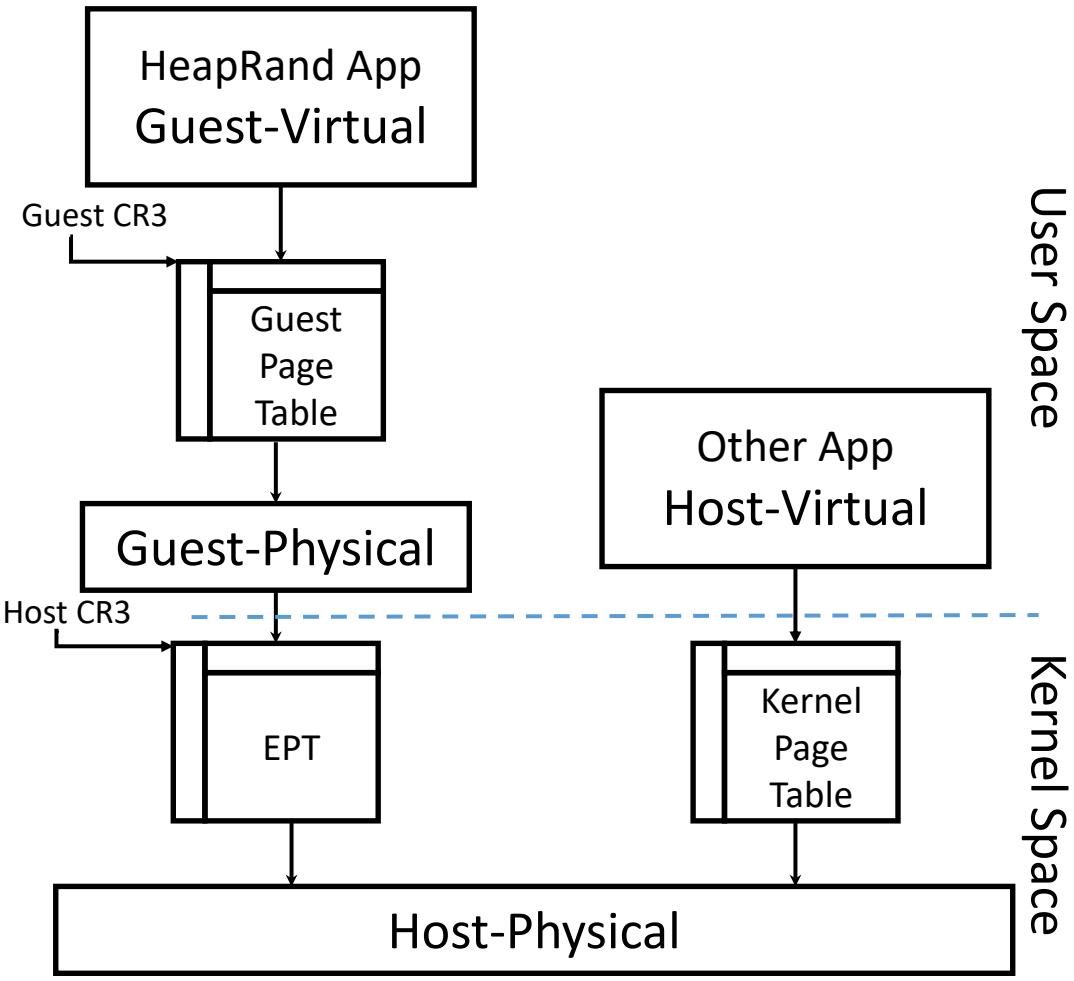


Figure 5.6 Address translation using Dune In Dune processes, the guest page table maps guest-virtual address to guest-physical address and the EPT that is managed by the kernel performs the additional translation to host-physical. Other applications used the kernel's standard page table

the program in each variant are guaranteed to be the same, except for pointers. Our design, in general, makes the following distinction for different types of data resident in memory:

- *Data*: This includes data used during the execution of the program, such as non-buffer variables resident in stack and heap, non-pointer variables in data and .bss regions of memory. Data is supposed to be the same across all variants at synchronization points.

- *Code/Data Pointers:* This includes code pointers, such as return addresses and function pointers as well as data pointers residing in stack, heap, data and .bss sections of the program memory. Based on the diversification introduced in variants by the MVX system, these could be different among variants. So, when the monitor encounters divergent data in the memory, it is assumed to be a code/data pointer. Based on the

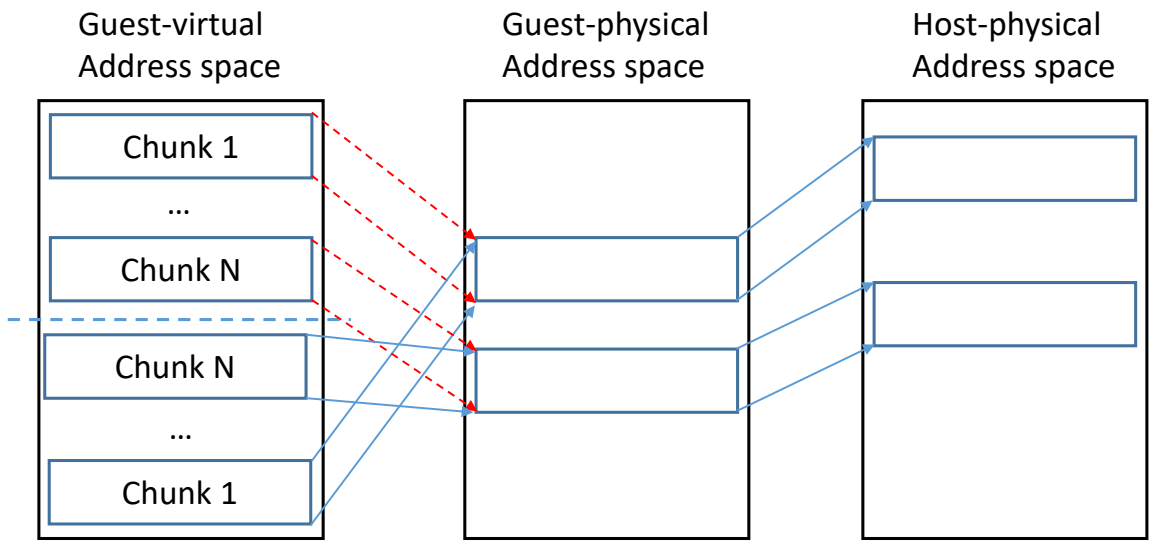


Figure 5.7 Re-randomizing BIBOP chunks with `HeapRand`

region of the memory a particular pointer points to, it could be identified as a pointer to a particular region, such as code, stack and heap.

Randomizing Chunks

`HeapRand` randomizes the address of a chunk used for a particular allocation class by mapping a new page and updating its guest page table of the guest virtual address to point to the guest physical address of the old page. `HeapRand` also needs to update the hash table used by the allocator to add the guest virtual address to map to the metadata address of the old bag, removing the entry of the old bag. `HeapRand` is suited to be used with memory allocators that keep mapping of pages to their corresponding metadata in a hash table to facilitate updates. To use `HeapRand` with allocators that use random offset would require additional metadata to facilitate the metadata update during re-randomization.

When any memory management function (such as malloc and free) is invoked the heap allocator performs the allocation and traps to the monitor to update the metadata. This metadata is required during the randomization to update pointers to heap allocations resident in various writable parts of memory, including stack, heap, data and .bss sections. Maintaining the metadata is particularly important to perform Randomization of pointers to heap objects that are resident in the heap itself, since the address of heap objects varies among variants. The monitor also updates the metadata used by the allocator.

Randomization Frequency

From a security perspective, it is essential to have a high re-randomization frequency, which limits the window an attacker has to synthesize and deploy a successful attack. This can be achieved by performing periodic re-randomization, in which the frequency of re-randomization is a trade-off with the incurred performance overhead. In our design, we make the distinction that any memory disclosure attempt by an attacker would require performing an output system call. Hence, we based our design on performing re-randomization at the granularity of output system calls.

Updating Pointers

Once the page table entries of the new chunks are updated, the next step is to find all pointers in the address space of the program, updating the pointers to allocations to the old chunk address to point to the new chunk address. `HeapRand` does this by comparing equivalent memory regions from both variants, including stack, heap, etc. If the memory layout of the variants is randomized, equivalent values show a non-pointer data, while a differing value shows a pointer. This requires the `HeapRand` monitor of the leading variant to access the address space of the follower. There are a number of options to access another process's address space:

- `ptrace` API: Memory reads using the ptrace API need to be performed in a 4-byte granularity, with each requiring a separate system call. Due to this requirement, it is very slow to be used for our purpose.

- Shared Memory: Shared memory APIs allow multiple processes virtual address pages to be mapped to the same physical address. Although shared memories have significant setting up cost, they provide a native like performance for the accesses there after. Even though it is an ideal choice for continual regions of memory (for e.g., stack), it will incur frequent setup cost for non-continual regions of memory (for e.g., the heap). Due to this, it is not an ideal choice for our purpose.

- `process_vm_read/write`: These system calls are introduced in Linux kenel 3.2 and allow transfer of a chunk of data between process address spaces without requiring the remote process to be stopped.

`process_vm_read/write` offers an efficient means to access the follower's address space. Section 5.5.1 shows the comparative latency of inter-process communication APIs. By comparing equivalent regions of memory in the leader variant and the follower variant

the monitor first identifies pointer values. If a heap pointer is found, its page address is used to locate the corresponding page virtual address of the new chunk, which is thus updated accordingly. Once all live memory is scanned and updated, the control is transferred back to the variants.

5.4 Implementation

We implemented `HeapRand` on top of Dune sandbox [113]. Dune enables virtualization of processes by safely exposing privileged features to a user level process. Even though `HeapRand` can be used with any BIBOP based allocator that maintains hash table for pages to metadata mapping, our prototype implementation uses the port of OpenBSD's allocator to Linux [114]. All allocations over 2KB are treated as large objects by OpenBSD, in which their allocation is performed using `mmap` operation instead. For our prototype implementation, we don't consider re-randomization of large allocations, as it would require non-trivial modification to the allocator. In addition, re-randomizing small allocations (*i.e.*, all allocation sized less than 2KB) will effectively change their relative distance with the large allocations.

Re-Mapping Chunks Upon output system calls, `HeapRand`'s monitor will block all variants and re-map all chunks that are alive based on the metadata stored during the allocation. During the re-mapping, the monitor maintains a hashmap for mapping the old guest-virtual address of a chunk to its new guest-virtual address. We keep the hashmap in a shared memory region and is accessed by all monitors.

To facilitate tracking of pointers, in our prototype implementation we kept all regions of the variants' memory except the heap at the same address, since `HeapRand` is only concerned with heap pointers. This is not a strict requirement for `HeapRand`, as a pointer to any region can be identified by adding extra checks to boundaries of the region. The monitor of the leader variant compares all memory regions that can occupy heap pointers, including the heap itself; based on the variation in values at equivalent addresses, the monitor identifies a heap pointer. If it is a heap pointer, the monitor updates its page address with the new address of the chunk.

Once all pointers are updated, we use `dune_flush_tlb_one(<addr>)` function to invalidate the TLB entries of the old allocation. As `HeapRand` utilizes a small number of pages at time, this function enables to flush only the required pages instead of the entire TLB. Belay et al. have demonstrated the advantage of this feature to improve the performance of

Table 5.1 Inter-process memory access Latency. This table shows the average number of clock cycle latency to read a memory page for each the process communication technique we evaluated and there normalized latency with an intra-process copy.

API	Latency (cycles/page)	Normalized
`Intra-process`	35.32	1
`Ptrace API`	1520.075	43.03
`Shared Memory`	38.58	1.09
`process_vm_read/write`	38.27	1.08

Boehm garbage collector [115].

5.5 Evaluation

In this section, we present the evaluation of `HeapRand`. We performed our experiments on a server based machine with Intel Xeon E5-2630 CPU with 32 cores and 128 GB RAM. Hyper-threading was disabled to reduce the performance fluctuations of the experimental results. All the experiments were run on a Red Hat Enterprise Linux 7.5 system, running a Linux kernel 3.10 (x86 64). We evaluated `HeapRand` on the SPEC CPU2006 benchmark suite as well as other network facing programs, focusing on accessing its performance overheads and its security guarantees.

5.5.1 Inter-process Memory Access Latency

We evaluated the run-time overhead induced by existing APIs for accessing memory across processes.We designed a simple microbenchmark that maps a large file (8 GB) in one process and which is read from another process in page size granularity using ptrace, POSIX shared memory and process_vm_read/write APIs. We took a geometric mean of 5 runs. Table 5.1 shows the latency incurred by each API and their normalized performance compared to a local memory access. The result shows that POSIX shared memory and `process_vm_read/write` APIs are capable of providing a native like performance for inter-process memory access, while ptrace API incurs significant orders of magnitude overhead because of its fixed (4 bytes) access granularity.

5.5.2 Performance Overhead

To assess the performance overhead, we evaluated the impact induced by `HeapRand` on SPEC 2006 benchmarks. Since we used OpenBSD allocator for each variant in `HeapRand`, we used the same allocator for the baseline runs. This helps to assess the overhead of `HeapRand` , which is independent from the overhead induced by allocator. Figure 5.8 shows the normalized performance overhead incurred by the MVX system alone and `HeapRand` at four different number of out system call granularities. As shown in the figure, the MVX system alone incurs an average overhead of 11%, while re-randomizing at every output system call incurs 32%. On benchmarks that have regular usage of the heap allocator and output system calls, `HeapRand` incurs moderate performance overheads. `HeapRand` incurs significant overhead on that put too much stresses on the heap allocator, such as perlbench and omnetpp. Table 5.3 shows the heap allocation statistics for reference inputs we used to evaluate each SPEC CPU2006 benchmark. In addition, this benchmarks perform a lot of output system calls, which induces a significant number of randomization during their runtime. While on certain memory intensive benchmarks (e.g., mcf and libquantum) the overhead is mostly a result of using MVX systems. This is consistent with the results observed in prior works, illustrating the poor scalability of these benchmarks with concurrent runs.

The performance results also show that benchmarks with a high number of output system calls respond well for increasing the granularity of the number of system calls between re-randomization. It also comes at no surprise that benchmarks with minimal output system call interactions exhibit similar behaviors for variation in the granularity of re-randomization. As it requires multiple memory disclosures to perform a successful attack [21], this property gives a performance trade-off for `HeapRand` while gracefully reducing security guarantees.

To evaluate the memory overhead of `HeapRand` we measured the amount of physical memory required to hold the running program's working set. Table 5.2 shows the maximum resident set size for running `HeapRand` on SPEC 2006 benchmarks. For most of the benchmarks we ran, the increase in the maximum resident set size is minimal. `hmmer` is an outlier, which showed 15x increase in the maximum resident set size, due to its unusually high number of reallocation operations for each allocation.

Table 5.3 shows the heap memory management function call statistics for the reference inputs we ran on each SPEC CPU2006 benchmark.

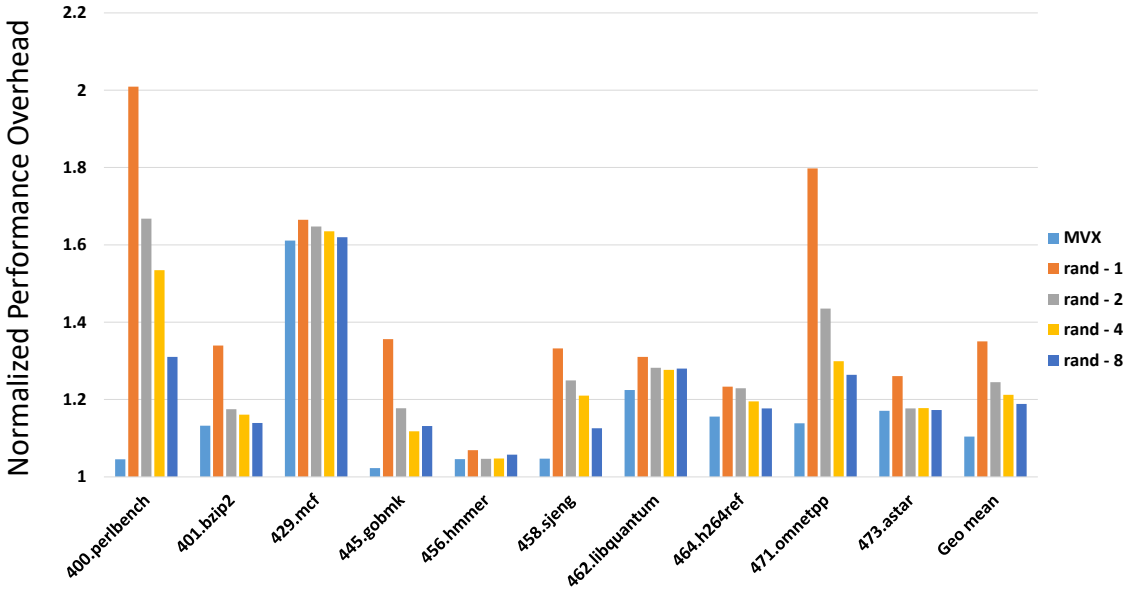


Figure 5.8 Normalized performance overhead This graph shows the performance overhead of running SPEC benchmarks on `HeapRand` at various granularities normalized to native runs. `MVX` shows the performance overhead incurred by the MVX system without any re-randomization, while `rand-1`, `rand-2`, `rand-4` and `rand-8` show re-randomization performed after 1, 2, 4 and 8 output system calls, respectively.

Table 5.2 Memory Overhead of HeapRand on SPEC benchmarks.

Benchmark	Native (KB)	MVX (KB)	HeapRand (KB)
`400.perlbench`	239036	246000	297160
`401.bzip2`	318532	325204	325228
`429.mcf`	1717020	1726380	1726400
`445.gobmk`	28892	36080	42040
`456.hmmer`	3892	13212	58716
`458.sjeng`	179508	186240	186260
`462.libquantum`	99084	108412	108432
`464.h264ref`	27876	35220	35432
`471.omnetpp`	173316	180564	203756
`473.astar`	334924	349956	394232

Table 5.3 Memory Management Functions call statistics for SPEC benchmarks.

Benchmark	malloc	calloc	realloc
400.perlbench	55095483	654	3863517
401.bzip2	28	1	0
429.mcf	2	4	0
445.gobmk	128282	0	8994
456.hmmer	1963593	122440	368134
458.sjeng	5	0	0
462.libquantum	0	122	58
464.h264ref	3196	35076	0
471.omnetpp	266808014	3705	1147
473.astar	3683333	15	16

5.5.3 Limitations

Our implementation `HeapRand` does not support multi-threaded applications since Dune's implementation of page table management is not thread safe. The `dune_flush_tlb_one()` function uses INVLPG instruction, which on some processors we tested invalidates the entire TLB instead of the particular TLB entry for the page that contains `addr`. Under this circumstances, `HeapRand` may induce a much higher performance degradation. This could be an implementation specific side-effect on certain Intel processors processors, as described in the the Intel ISA reference manual. [3]

5.6 Related Work

Similar to stack data protections, heap protections can be categorized as enforcement based and randomization based protection.

Enforcement Based Protections. Complete memory safety techniques that enforce spatial and temporal memory safety, such as Softbound+CETs [27], fat pointers [26], and low-fat pointers [28, 117], fall in this category. However, complete memory safety techniques are infeasible for production systems because of their significantly high overhead. Other enforcement based heap protections have been proposed that append random canary words to each heap allocated object and check integrity of all the registered canaries every

[3]"There are implementation-specific side effect on the Pentium 4, Intel Xeon, and P6 processor family. When modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, which includes global entries. Software implementations should take this into consideration when using this functionality in the above Intel 64 or IA-32 processors." [116]

time the process requests a system call [118]. Even though heap canaries incur a relatively lower overhead, they only protect against sequential heap overflows.

Randomization Based Protections. Sequential allocators, such as DLmalloc and Windows, maintain a linked list of free allocations for different size classes to assist fast allocations. This metadata is stored adjacent to the allocated buffers and is prone to overwrites. Randomization based heap protections work by adding entropy to the allocation layout, the reuse of freed allocations and the location of allocation metadata to mitigate one or more of the attacks discussed in section 5.2.1. OpenBSD's allocator [119] segregates the heap metadata from the heap allocations, uses sparse page layout, and introduces 4 bits of randomization to new allocations as well as reuse of freed allocations. Like OpenBSD, DieHard [120] randomizes the placement of allocated objects and the length of time before freed objects are recycled. However, unlike OpenBSD's random choice from 16 entries, DieHard picks one out of all the available allocations of given sizes through a bitmap based metadata. In addition, DieHard uses a replicated execution framework to improve reliability. However, DieHard doesn't use a sparse page layout, which increases the exploitability of overflows. Furthermore, its reliability feature enables the program to continue running after experiencing memory errors, without being detected. DieHarder [95] improves the limitations of DieHard by using OpenBSD's sparse page layout, destroying on free features, and reporting violations, instead of continuing operations upon errors. Bitmap based allocators incur high overhead compared to sequential allocators; for example, DieHarder results in an average slowdown of 36% on SPEC2006 benchmarks over DLmalloc. More recent heap layout randomization schemes [121][122] adopt the inline freelist idea from sequential allocators to improve performance and utilize sparse pages and randomize the layout of the allocation like secure allocators. Furthermore, there are heap data protections which are tailored to a particular type heap corruption exploit, such as use-after free [123].

Generally randomization based heap allocators offer a probabilistic security guarantee and have relatively less overhead than enforcement based techniques. Due to these capabilities, they are more suited for production environments than enforcement techniques. One typical example is OpenBSD's allocator.

In this work we propose re-randomization of heap resident buffers during the runtime of the program.

5.7 Chapter Summary

With an increase in number and sophistication of control-flow hijacking attack protections, data only attacks have become more popular for penetrating runtime systems. These attacks bypass control-flow integrity techniques, as they do not alter any control flow data whose integrity is enforced by these protections. The most important distinction of data plane attacks is that they utilize relative distance between memory regions instead of the absolute memory address, which is a requirement for most control flow based attacks. Secure heap allocators randomize the relative distance between allocation by randomly choosing an allocations from multiple possible allocation class chunks as well as randomzing the reuse of freed allocations. These ensures a randomized distance among heap objects during allocation. However, the relative distance between allocations is intact until the allocation is freed. The relative distance can be disclosed considering the abundance of memory leak attacks.

In this work we introduce `HeapRand`, a technique to randomize the relative distance between heap allocation in order to mitigate disclosure of relative distance among allocations, which can be utilized to perform a successful code-reuse exploit. `HeapRand` uses Dune, which enables user-level access of privileged features by providing process virtualization, to remap pages used as allocation chunks. By performing remapping of chunks, `HeapRand` effectively randomizes the relative distance between heap allocations with moderate overhead. Our evaluation shows that `HeapRand` incurs moderate performance overhead.

Chapter 6
Conclusion and Future Direction

6.1 Dissertation Conclusion

Exploitation of most modern systems relies on the principle of code-reuse attacks, since they do not need injecting code in the system and are capable to bypass built-in defenses to perform arbitrary computation. Classical code-reuse attacks, such as ROP, are used to hijack the control-flow of the program. With an increase in wide spread adoption of low overhead control flow defences, there is proliferation of advanced code-reuse attacks that can exploit systems protected by these defences.

Most advanced code-reuse attacks utilize relative distance between memory allocations instead of absolute addresses, which is the asset protected by earlier defences. Even though enforcement based techniques provide a comprehensive solution to memory corruption vulnerabilities, they are hampered from being able to be deployed in production systems by their high overheads and backward incompatibility issues. Because of these reasons enforcement techniques are mostly used as debugging tools. Randomization techniques, on the other hand, introduce diversification in the program to provide probabilistic security guarantees with minimal overheads. The security guarantee afforded by these techniques can be tuned by the amount of diversification introduced. In this dissertation, we propose three techniques to thwart these advanced attacks.

In Chapter 3, we presented ProxyCFI, a CFI based technique to thwart code-reuse attacks including CFG mimicry attacks. Classic CFI techniques inserts instrument indirect branch instructions to check the validity of their target according to the application's legitimate control-flow graph. ProcyCFI instead totally eliminates indirect branches from the program by replacing them with white-listed multi-way direct branches that use pointer proxies. Our implementation of ProxyCFI instruments indirect control flow transfer instructions at compile time whose values are re-randomized at application load-time to prevent them from static analysis based discovery of pointer proxies. ProxyCFI also has a verifier built-in the program loader to ensure all binaries are checked for compliance before being loaded. Due

to these features, ProxyCFI puts a strict limit on advanced code-reuse attacks. With our optimizations applied ProxyCFI affords strong security guarantees with minimal performance overhead.

In chapter 4, we presented Smokestack, a binary hardening compiler that automatically translates a vulnerable program in to a self randomizing program that dynamically re-randomizes the layout of each stack frame for all function invocations. At the heart of Smokestack is the static analysis pass, which performs analysis of each stack frame in the program independently to capture all possible permutations of the allocations within it. These is fed to an instrumentation pass that instruments the prologue of all functions to pick a random permutation based on a secure random-number generated during the invocation. The experimental evaluation demonstrates that Smokestack achieves high effectiveness in stopping advanced code-reuse attacks while incurring reasonable performance overheads. As Smokestack is a technique to introduce artificial diversification it provides a robust protection to hardened programs in the face of memory leakage and repeated de-randomization attempts.

In chapter 5, we present HeapRand, a runtime heap layout randomization scheme. HeapRand identifies pointers to heap allocations by comparing the values of memory locations in two versions running with same memory layout except for the heap throughout their execution. HeapRand uses BIBOP based heap allocators for each variant with different randomization and performs randomization of addresses of chunks used for allocations at output system call granurality.

6.2 Future Direction

Through out this dissertation, we have identified advanced code-reuse attacks and how they may circumvent prior proposed defenses as well as defenses that are builtin modern systems. In chapter 3, we discussed prior implementations of CFI can be undermined by attacks that piggy back on the enforced CFG. In chapter 4 and 5, we showed how data-oriented attacks can bypass defenses by only utilizing relative distance among stack and heap allocation and presented our strategies to defend against these attacks. In this section, we will present the future research landscape in protections against advances in code-reuse attacks.

In Chapter 3, we presented ProxyCFI which randomizes code pointers at program load time. This requires compiling of source codes with ProxyCFI compiler. One important challenge for any CFI technique is extracting precise control flow graph of a program. As the strength of a CFI solution is directly tied with the precision of the enforced CFG,

precise CFG discovery is an active research area. In addition, ProxyCFI can be extended to support legacy binaries, by using binary analysis tools to generate the CFG of the program and doing binary rewriting to update all indirect branches to use pointer proxies instead. ProxyCFI currently doesn't support applications that run JITted code like browsers, as it is impossible to get the executed code during our static analysis. One viable solution to support JITted code in ProxyCFI is to modify system such as `mprotect()`, to detect calls during JIT compilation in order to instrument the JITted code to use pointer proxies at runtime. Generally, the effectiveness of any CFI technique depends on discovering a precise CFG of the program, which is a topic of utmost interest. The use of advanced pointer analyses, such as Multi-layer type analysis schemes, can significantly reduce the attack surface for CFG mimicry attacks.

Smokestack introduces a stack layout randomization technique to defend against code-reuse attack that rely on relative offset between allocations. Current implementation of Smokestack, however, has certain limitations. First, Smokestack can not randomize intra-record allocations. One possible solution for this is to adopt the data structure randomization scheme proposed by Peng et al [124]. Another weakness of Smokestack is that the randomization entropy is dependent on the number of allocations in a stack frame. For example, if a stack frame has just 2 allocations, the entropy introduced by Smokestack is just 50%. Smokestack addresses this issue by adding random sized puddings in the allocations with having trade off of using extra memory space. However, it still doesn't give equal security guarantees for functions with different sized and number of allocations. This is a common problem in randomization based allocations [112]. One possible solution for this problem is the use of over-provisioning similar to Dieharder [95], to provide a constant minimum security guarantee.

HeapRand uses dune based MVX system to randomize heap resident allocations. As MVX systems have the potential to provide a modular support to adopt more defenses, it can be extended to randomize other regions of memory of memory such as stack, data and .bss regions. Current implementation of HeapRand keeps other regions of memory, including stack and code, at the same location across variants to facilitate identifying heap allocation pointers from pointers to other regions of memory. This can weaken the security guarantee provided by MVX for other regions of memory. A possible solution for this issue is to keep bounds of regions of memory in the monitor and identify pointers to a particular region based on whether its value maps to which region.

This thesis illustrates that randomization based defenses (*i.e.*, avoiding discoverability of assets used for code-reuse attacks) and re-randomization (*i.e.*, putting a strict time limit to perform an attack) are favorable to stop advances in code-reuse attack. The effectiveness of

randomization based defenses in defending against code-reuse attacks hinges on the amount of the entropy the schemes provide. Most proposed defenses provide limited entropy in practice to reduce performance overheads, which in turn makes them vulnerable to more sophisticated attacks (e.g., side channels [125], guessing [126], spraying [4]) or disclosure attacks [21]. If the entropy source is compromised, an adversary will be able to predict or even control the re-randomization to effectively bypass it. A promising approach towards improved entropy is to provide a hardware support for true random number generation and strong cipher algorithms [127][128][129].

Bibliography

[1] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[2] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[3] w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999. Accessed: 2018-04-13.

[4] Heap spraying buffer overflow attacks. https://tools.cisco.com/security/center/viewAlert.x?alertId=21136, 2010. Accessed: 2018-04-13.

[5] Windows isv software security defenses, 2010. Accessed: 2018-02-29.

[6] Linux kernel 2.6.8, 2004. Accessed: 2018-02-29.

[7] Data execution prevention, 2003. Accessed: 2018-02-29.

[8] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[9] Dino Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.

[10] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[11] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[12] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[13] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[14] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951. ACM, 2015.

[15] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[16] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.

[17] Yu Yang. Rops are for the 99%, cansecwest 2014. *h ttps://cansecwest. com/slides/2014/ROPs are for the*, 99, 2014.

[18] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009.

[19] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[20] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[21] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[22] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[23] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet denial of service: attack and defense mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR, 2004.

[24] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[25] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-oriented programming: a new code reuse attack to bypass modern defenses. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 190–197. IEEE, 2015.

[26] Todd M Austin, Scott E Breach, and Gurindar S Sohi. *Efficient detection of all pointer and array access errors*, volume 29. ACM, 1994.

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[28] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Network and Distributed System Security Symposium (NDSS) 2017*, 2017.

[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[30] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. *ACM Sigplan Notices*, 45(8):31–40, 2010.

[32] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[33] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, 2014. USENIX Association.

[34] Miguel Castro, Manuel Costa, and Timothy L. Harris. Securing software by enforcing data-flow integrity. In *OSDI*, 2006.

[35] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277. IEEE, 2008.

[36] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[37] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[38] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 299–310. ACM, 2013.

[39] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[40] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[41] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[42] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[43] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[44] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 469–484. ACM, 2019.

[45] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[46] Once upon a free(). http://phrack.org/issues/57/9.html, 2001. Accessed: 2018-04-13.

[47] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

[48] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 37–44. IEEE, 2011.

[49] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[50] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, pages 337–352, 2013.

[51] Intel control-flow enforcement technology (cet). https://software.intel.com/sites/default/files/managed /4d/2a/control-flow-enforcement-technology-preview.pdf, 2017. Accessed: 2018-04-13.

[52] Control flow guard (windows) - msdn - microsoft. https://msdn.microsoft.com/en-us/library/dn919635.aspx, 2015. Accessed: 2018-04-13.

[53] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014.

[54] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913. ACM, 2015.

[55] Misiker Tadesse Aga, Colton Holoday, and Todd Austin. Wrangling in the power of code pointers with proxycfi. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 317–337. Springer, 2019.

[56] Michael Theodorides and David Wagner. Breaking active-set backward-edge CFI. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*, pages 85–89. IEEE, 2017.

[57] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[58] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[59] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[60] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 417–432. USENIX Association, 2014.

[61] Intel. Dynamic libraries, 2015. Accessed 2018-02-29.

[62] Keith D Cooper, Mary W Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.

[63] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

[64] Cve-2014-2013. https://www.cvedetails.com/cve/ CVE-2014-2013/, 2018. Accessed: 2018-04-13.

[65] Bladeenc: Vulnerability statistics. https://www.cvedetails.com/product/2851/Bladeenc-Bladeenc.html, 2018. Accessed: 2018-01-05.

[66] Cve-2017-14493. https://www.cvedetails.com/cve/ CVE -2017-14493/, 2017. Accessed: 2018-02-12.

[67] Cve-2017-1000437. https://www.cvedetails.com/cve/ CVE-2017-1000437/, 2018. Accessed: 2018-01-05.

[68] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015.

[69] National vulnerability database. Accessed: 2018-02-29.

[70] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. Getting in control of your control flow with control-data isolation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 79–90. IEEE Computer Society, 2015.

[71] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[72] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[73] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, and Dawn Song. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. In *IEEE Symposium on Security and Privacy*, 2015.

[74] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*, volume 14, page 00000, 2014.

[75] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[76] Cve-2018-1000140, 2018. Accessed: 2018-05-14.

[77] Misiker Tadesse Aga and Todd Austin. Smokestack: thwarting dop attacks with runtime stack layout randomization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 26–36. IEEE, 2019.

[78] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[79] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[80] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[81] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72. IEEE, 1997.

[82] Shay Gueron. Intel advanced encryption standard (aes) instruction set white paper, rev, 2010.

[83] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[84] Oprofile, 2018. Accessed: 2018-05-10.

[85] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. Ripe: runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2011.

[86] Stack-based buffer overflow in the mpeg parser in wireshark, 2015. Accessed: 2018-03-14.

[87] Stack-based buffer overflow in the sreplace function in proftpd 1.3.0, 2006. Accessed: 2018-03-14.

[88] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[89] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

[90] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[91] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, pages 367–382, 2016.

[92] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. Hardscope: Thwarting dop with hardware-assisted run-time scope enforcement. *arXiv preprint arXiv:1705.10295*, 2017.

[93] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 2000–02, 2000.

[94] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[95] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[96] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged {CPU} features. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 335–348, 2012.

[97] Memory errors. Accessed: 2018-12-06.

[98] Advancing exploitation: a scriptless 0day exploit against linux desktops, 2018. Accessed: 2018-11-27.

[99] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[100] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[101] Lorenzo Cavallaro. *Comprehensive memory error protection via diversity and taint-tracking*. PhD thesis, PhD thesis, PhD dissertation, Universita Degli Studi Di Milano, 2007.

[102] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 612–621. IEEE Press, 2013.

[103] Matthew Maurer and David Brumley. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*. USENIX, 2012.

[104] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46. ACM, 2009.

[105] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. Ghumvee: efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security*, pages 261–277. Springer, 2012.

[106] Jack W Davidson, Jason D Hiser, John C Knight, Anh Nguyen-Tuong, Westley Weimer, Jonathan Burket, Gregory L Frazier, Tiffany M Frazier, Bruno Dutertre, Ian Mason, et al. Double helix and raven: A system for cyber fault tolerance and recovery. In *11th Annual Cyber and Information Security Research Conference, CISRC 2016*, page 2897805. Association for Computing Machinery, Inc, 2016.

[107] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, pages 105–120, 2006.

[108] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *USENIX Annual Technical Conference*, pages 167–179, 2016.

[109] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[110] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 339–353. ACM, 2015.

[111] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 329–343. ACM, 2017.

[112] Otto Moerbeek. A new malloc (3) for openbsd. In *Proceedings of the 2009 European BSD Conference, EuroBSDCon*, volume 9, 2009.

[113] The dune project. https://github.com/emeryberger/Malloc-Implementations, 2012.

[114] Malloc implementations. https://github.com/project-dune/dune, 2012.

[115] Hans-J Boehm, Alan J Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI*, volume 91, pages 157–164. Citeseer, 1991.

[116] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[117] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[118] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: kernel-assisted protection against heap overflows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2013.

[119] Otto Moerbeek. A new malloc (3) for openbsd. In *Proceedings of the 2009 European BSD Conference, EuroBSDCon*, volume 9, 2009.

[120] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[121] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403. ACM, 2017.

[122] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 117–133, 2018.

[123] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[124] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. A practical approach for adaptive data structure layout randomization. In *European Symposium on Research in Computer Security*, pages 69–89. Springer, 2015.

[125] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016.

[126] Yoav Weiss and Elena Gabriela Barrantes. Known/chosen key attacks against software instruction set randomization. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 349–360. IEEE, 2006.

[127] JV Roig. Siderand: A heuristic and prototype of a side-channel-based cryptographically secure random seeder designed to be platform-and architecture-agnostic. *arXiv preprint arXiv:1804.02904*, 2018.

[128] DJ Bernstein. Fast-key-erasure random-number-generators (2017). *Available: blog. cr. yp. to/20170723-random. html*, 2017.

[129] D Eastlake, J Schiller, and Steve Crocker. Randomness requirements for security. *RFC4086*, 2005.